The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.9

Andrew Waterman
Yunsup Lee
Rimas Avizienis
David A. Patterson
Krste Asanović

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2016-129
http://www.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-129.html

July 8, 2016

The RISC-V Instruction Set Manual

Volume II: Privileged Architecture

Privileged Architecture Version 1.9

Document Version 1.9

Warning! This draft specification will change before being accepted as standard, so implementations made to this draft specification will likely not conform to the future standard.

Andrew Waterman, Yunsup Lee, Rimas Avižienis, David Patterson, Krste Asanović
CS Division, EECS Department, University of California, Berkeley
{waterman|yunsup|rimas|pattrsn|krste}@eecs.berkeley.edu

July 8, 2016

Contents

Chapter 1

Introduction

This is a draft of the privileged architecture description document for RISC-V. Feedback welcome. Changes will occur before the final release.

This document describes the RISC-V privileged architecture, which covers all aspects of RISC-V systems beyond the user-level ISA, including privileged instructions as well as additional functionality required for running operating systems and attaching external devices.

> *Commentary on our design decisions is formatted as in this paragraph, and can be skipped if the reader is only interested in the specification itself.*

> *We briefly note that the entire privileged-level design described in this document could be replaced with an entirely different privileged-level design without changing the user-level ISA, and possibly without even changing the ABI. In particular, this privileged specification was designed to run existing popular operating systems, and so embodies the conventional level-based protection model. Alternate privileged specifications could embody other more flexible protection domain models.*

1.1 RISC-V Hardware Platform Terminology

A RISC-V hardware platform can contain one or more RISC-V-compatible processing cores together with other non-RISC-V-compatible cores, fixed-function accelerators, various physical memory structures, I/O devices, and an interconnect structure to allow the components to communicate.

A component is termed a *core* if it contains an independent instruction fetch unit. A RISC-V-compatible core might support multiple RISC-V-compatible hardware threads, or *harts*, through multithreading.

A RISC-V core might have additional specialized instruction set extensions or an added *coprocessor*. We use the term *coprocessor* to refer to a unit that is attached to a RISC-V core and is mostly sequenced by a RISC-V instruction stream, but which contains additional architectural state and instruction set extensions, and possibly some limited autonomy relative to the primary RISC-V instruction stream.

We use the term *accelerator* to refer to either a non-programmable fixed-function unit or a core that can operate autonomously but is specialized for certain tasks. In RISC-V systems, we expect many programmable accelerators will be RISC-V-based cores with specialized instruction set extensions and/or customized coprocessors. An important class of RISC-V accelerators are I/O accelerators, which offload I/O processing tasks from the main application cores.

The system-level organization of a RISC-V hardware platform can range from a single-core microcontroller to a many-thousand-node cluster of shared-memory manycore server nodes. Even small systems-on-a-chip might be structured as a hierarchy of multicomputers and/or multiprocessors to modularize development effort or to provide secure isolation between subsystems.

This document focuses on the privileged architecture visible to each hart (hardware thread) running within a uniprocessor or a shared-memory multiprocessor.

1.2 RISC-V Privileged Software Stack Terminology

This section describes the terminology we use to describe components of the wide range of possible privileged software stacks for RISC-V.

Figure 1.1 shows some of the possible software stacks that can be supported by the RISC-V architecture. The left-hand side shows a simple system that supports only a single application running on an application execution environment (AEE). The application is coded to run with a particular application binary interface (ABI). The ABI includes the supported user-level ISA plus a set of ABI calls to interact with the AEE. The ABI hides details of the AEE from the application to allow greater flexibility in implementing the AEE. The same ABI could be implemented natively on multiple different host OSs, or could be supported by a user-mode emulation environment running on a machine with a different native ISA.

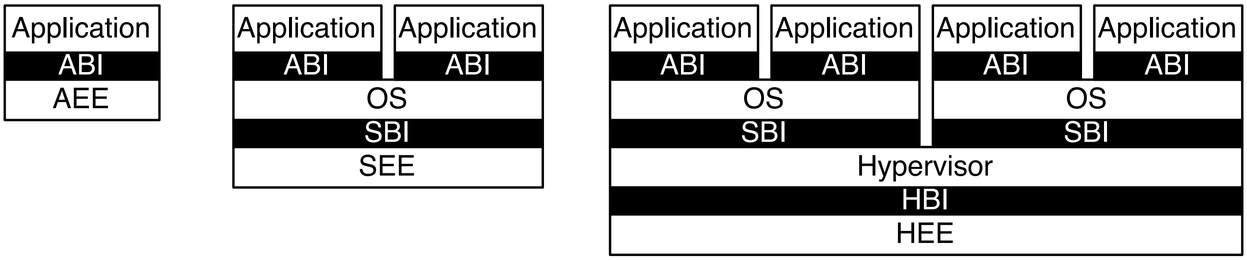

Figure 1.1: Different implementation stacks supporting various forms of privileged execution.

Our graphical convention represents abstract interfaces using black boxes with white text, to separate them from concrete instances of components implementing the interfaces.

The middle configuration shows a conventional operating system (OS) that can support multiprogrammed execution of multiple applications. Each application communicates over an ABI with the OS, which provides the AEE. Just as applications interface with an AEE via an ABI, RISC-V operating systems interface with a supervisor execution environment (SEE) via a supervisor binary interface (SBI). An SBI comprises the user-level and supervisor-level ISA together with a set of

SBI function calls. Using a single SBI across all SEE implementations allows a single OS binary image to run on any SEE. The SEE can be a simple boot loader and BIOS-style IO system in a low-end hardware platform, or a hypervisor-provided virtual machine in a high-end server, or a thin translation layer over a host operating system in an architecture simulation environment.

Most supervisor-level ISA definitions do not separate the SBI from the execution environment and/or the hardware platform, complicating virtualization and bring-up of new hardware platforms.

The rightmost configuration shows a virtual machine monitor configuration where multiple multiprogrammed OSs are supported by a single hypervisor. Each OS communicates via an SBI with the hypervisor, which provides the SEE. The hypervisor communicates with the hypervisor execution environment (HEE) using a hypervisor binary interface (HBI), to isolate the hypervisor from details of the hardware platform.

The various ABI, SBI, and HBIs are still a work-in-progress, but we anticipate the SBI and HBI to support devices via virtualized device interfaces similar to virtio [3], and to support device discovery. In this manner, only one set of device drivers need be written that can support any OS or hypervisor, and which can also be shared with the boot environment.

Hardware implementations of the RISC-V ISA will generally require additional features beyond the privileged ISA to support the various execution environments (AEE, SEE, or HEE).

1.3 Privilege Levels

At any time, a RISC-V hardware thread (*hart*) is running at some privilege level encoded as a mode in one or more CSRs (control and status registers). Four RISC-V privilege levels are currently defined as shown in Table 1.1.

Level	Encoding	Name	Abbreviation
0	`00`	User/Application	U
1	`01`	Supervisor	S
2	`10`	Hypervisor	H
3	`11`	Machine	M

Table 1.1: RISC-V privilege levels.

Privilege levels are used to provide protection between different components of the software stack, and attempts to perform operations not permitted by the current privilege mode will cause an exception to be raised. These exceptions will normally cause traps into an underlying execution environment or the HAL.

The machine level has the highest privileges and is the only mandatory privilege level for a RISC-V hardware platform. Code run in machine-mode (M-mode) is inherently trusted, as it has low-level access to the machine implementation. M-mode is used to manage secure execution environments on RISC-V. User-mode (U-mode) and supervisor-mode (S-mode) are intended for conventional application and operating system usage respectively, while hypervisor-mode (H-mode) is intended to support virtual machine monitors.

Each privilege level has a core set of privileged ISA extensions with optional extensions and variants. For example, machine-mode supports several optional standard variants for address translation and memory protection.

Although none are currently defined, future hypervisor-level ISA extensions will be added to improve virtualization performance. One common feature to support hypervisors is to provide a second level of translation and protection, from supervisor physical addresses *to* hypervisor physical addresses*.*

Implementations might provide anywhere from 1 to 4 privilege modes trading off reduced isolation for lower implementation cost, as shown in Table 1.2.

In the description, we try to separate the privilege level *for which code is written, from the* privilege mode *in which it runs, although the two are often tied. For example, a supervisor-level operating system can run in supervisor-mode on a system with three privilege modes, but can also run in user-mode under a classic virtual machine monitor on systems with two or more privilege modes. In both cases, the same supervisor-level operating system binary code can be used, coded to a supervisor-level SBI and hence expecting to be able to use supervisor-level privileged instructions and CSRs. When running a guest OS in user mode, all supervisor-level actions will be trapped and emulated by the SEE running in the higher-privilege level.*

Number of levels	Supported Modes	Intended Usage
1	M	Simple embedded systems
2	M, U	Secure embedded systems
3	M, S, U	Systems running Unix-like operating systems
4	M, H, S, U	Systems running Type-1 hypervisors

Table 1.2: Supported combinations of privilege modes.

All hardware implementations must provide M-mode, as this is the only mode that has unfettered access to the whole machine. The simplest RISC-V implementations may provide only M-mode, though this will provide no protection against incorrect or malicious application code. Many RISC-V implementations will also support at least user mode (U-mode) to protect the rest of the system from application code. Supervisor mode (S-mode) can be added to provide isolation between a supervisor-level operating system and the SEE and HAL code. The hypervisor mode (H-mode) is intended to provide isolation between a virtual machine monitor and a HEE and HAL running in machine mode.

A hart normally runs application code in U-mode until some trap (e.g., a supervisor call or a timer interrupt) forces a switch to a trap handler, which usually runs in a more privileged mode. The hart will then execute the trap handler, which will eventually resume execution at or after the original trapped instruction in U-mode. Traps that increase privilege level are termed *vertical* traps, while traps that remain at the same privilege level are termed *horizontal* traps. The RISC-V privileged architecture provides flexible routing of traps to different privilege layers.

Horizontal traps can be implemented as vertical traps that return control to a horizontal trap handler in the less-privileged mode.

Chapter 2

Control and Status Registers (CSRs)

The SYSTEM major opcode is used to encode all privileged instructions in the RISC-V ISA. These can be divided into two main classes: those that atomically read-modify-write control and status registers (CSRs), and all other privileged instructions. In addition to the user-level state described in Volume I of this manual, an implementation may contain additional CSRs, accessible by some subset of the privilege levels using the CSR instructions described in the user-level manual. In this chapter, we map out the CSR address space. The following chapters describe the function of each of the CSRs according to privilege level, as well as the other privileged instructions which are generally closely associated with a particular privilege level. Note that although CSRs and instructions are associated with one privilege level, they are also accessible at all higher privilege levels.

2.1 CSR Address Mapping Conventions

The standard RISC-V ISA sets aside a 12-bit encoding space (csr[11:0]) for up to 4,096 CSRs. By convention, the upper 4 bits of the CSR address (csr[11:8]) are used to encode the read and write accessibility of the CSRs according to privilege level as shown in Table 2.1. The top two bits (csr[11:10]) indicate whether the register is read/write (`00`, `01`, or `10`) or read-only (`11`). The next two bits (csr[9:8]) encode the lowest privilege level that can access the CSR.

The CSR address convention uses the upper bits of the CSR address to encode default access privileges. This simplifies error checking in the hardware and provides a larger CSR space, but does constrain the mapping of CSRs into the address space.

Implementations might allow a more-privileged level to trap otherwise permitted CSR accesses by a less-privileged level to allow these accesses to be intercepted. This change should be transparent to the less-privileged software.

Attempts to access a non-existent CSR raise an illegal instruction exception. Attempts to access a CSR without appropriate privilege level or to write a read-only register also raise illegal instruction exceptions. A read/write register might also contain some bits that are read-only, in which case writes to the read-only bits are ignored.

CSR Address			Hex	Use and Accessibility
[11:10]	[9:8]	[7:6]		
User CSRs				
00	00	XX	0x000-0x0FF	Standard read/write
01	00	XX	0x400-0x4FF	Standard read/write
10	00	XX	0x800-0x8FF	Non-standard read/write
11	00	00-10	0xC00-0xCBF	Standard read-only
11	00	11	0xCC0-0xCFF	Non-standard read-only
Supervisor CSRs				
00	01	XX	0x100-0x1FF	Standard read/write
01	01	00-10	0x500-0x5BF	Standard read/write
01	01	11	0x5C0-0x5FF	Non-standard read/write
10	01	00-10	0x900-0x9BF	Standard read/write shadows
10	01	11	0x9C0-0x9FF	Non-standard read/write shadows
11	01	00-10	0xD00-0xDBF	Standard read-only
11	01	11	0xDC0-0xDFF	Non-standard read-only
Hypervisor CSRs				
00	10	XX	0x200-0x2FF	Standard read/write
01	10	00-10	0x600-0x6BF	Standard read/write
01	10	11	0x6C0-0x6FF	Non-standard read/write
10	10	00-10	0xA00-0xABF	Standard read/write shadows
10	10	11	0xAC0-0xAFF	Non-standard read/write shadows
11	10	00-10	0xE00-0xEBF	Standard read-only
11	10	11	0xEC0-0xEFF	Non-standard read-only
Machine CSRs				
00	11	XX	0x300-0x3FF	Standard read/write
01	11	00-10	0x700-0x79F	Standard read/write
01	11	10	0x7A0-0x7BF	Standard read/write debug CSRs
01	11	11	0x7C0-0x7FF	Non-standard read/write
10	11	00-10	0xB00-0xBBF	Standard read/write shadows
10	11	11	0xBC0-0xBFF	Non-standard read/write shadows
11	11	00-10	0xF00-0xFAF	Standard read-only
11	11	11	0xFC0-0xFFF	Non-standard read-only

Table 2.1: Allocation of RISC-V CSR address ranges.

Table 2.1 also indicates the convention to allocate CSR addresses between standard and non-standard uses. The CSR addresses reserved for non-standard uses will not be redefined by future standard extensions. The shadow addresses are reserved to provide a read-write address via which a higher privilege level can modify a register that is read-only at a lower privilege level. Note that if one privilege level has already allocated a read/write shadow address, then any higher privilege level can use the same CSR address for read/write access to the same register.

Effective virtualization requires that as many instructions run natively as possible inside a virtualized environment, while any privileged accesses trap to the virtual machine monitor [1]. CSRs that are read-only at some lower privilege level are shadowed into separate CSR addresses if they are made read-write at a higher privilege level. This avoids trapping permitted lower-privilege *accesses while still causing traps on illegal accesses.*

Machine-mode standard read-write CSRs `0xFA0`–`0xFBF` are reserved for use by the debug system. Implementations should raise illegal instruction exceptions on machine-mode access to these registers.

2.2 CSR Listing

Tables 2.2–2.6 lists the CSRs that have currently been allocated CSR addresses. The timers, counters, and floating-point CSRs are the only standard user-level CSRs currently defined. The other registers are used by privileged code, as described in the following chapters. Note that not all registers are required on all implementations.

Number	Privilege	Name	Description
			User Trap Setup
`0x000`	URW	`ustatus`	User status register.
`0x004`	URW	`uie`	User interrupt-enable register.
`0x005`	URW	`utvec`	User trap handler base address.
			User Trap Handling
`0x040`	URW	`uscratch`	Scratch register for user trap handlers.
`0x041`	URW	`uepc`	User exception program counter.
`0x042`	URW	`ucause`	User trap cause.
`0x043`	URW	`ubadaddr`	User bad address.
`0x044`	URW	`uip`	User interrupt pending.
			User Floating-Point CSRs
`0x001`	URW	`fflags`	Floating-Point Accrued Exceptions.
`0x002`	URW	`frm`	Floating-Point Dynamic Rounding Mode.
`0x003`	URW	`fcsr`	Floating-Point Control and Status Register (`frm` + `fflags`).
			User Counter/Timers
`0xC00`	URO	`cycle`	Cycle counter for RDCYCLE instruction.
`0xC01`	URO	`time`	Timer for RDTIME instruction.
`0xC02`	URO	`instret`	Instructions-retired counter for RDINSTRET instruction.
`0xC80`	URO	`cycleh`	Upper 32 bits of `cycle`, RV32I only.
`0xC81`	URO	`timeh`	Upper 32 bits of `time`, RV32I only.
`0xC82`	URO	`instreth`	Upper 32 bits of `instret`, RV32I only.

Table 2.2: Currently allocated RISC-V user-level CSR addresses.

Number	Privilege	Name	Description
Supervisor Trap Setup			
`0x100`	SRW	`sstatus`	Supervisor status register.
`0x102`	SRW	`sedeleg`	Supervisor exception delegation register.
`0x103`	SRW	`sideleg`	Supervisor interrupt delegation register.
`0x104`	SRW	`sie`	Supervisor interrupt-enable register.
`0x105`	SRW	`stvec`	Supervisor trap handler base address.
Supervisor Trap Handling			
`0x140`	SRW	`sscratch`	Scratch register for supervisor trap handlers.
`0x141`	SRW	`sepc`	Supervisor exception program counter.
`0x142`	SRW	`scause`	Supervisor trap cause.
`0x143`	SRW	`sbadaddr`	Supervisor bad address.
`0x144`	SRW	`sip`	Supervisor interrupt pending.
Supervisor Protection and Translation			
`0x180`	SRW	`sptbr`	Page-table base register.
Supervisor Counter/Timers			
`0xD00`	SRO	`scycle`	Supervisor cycle counter.
`0xD01`	SRO	`stime`	Supervisor wall-clock time.
`0xD02`	SRO	`sinstret`	Supervisor instructions-retired counter.
`0xD80`	SRO	`scycleh`	Upper 32 bits of `scycle`, RV32I only.
`0xD81`	SRO	`stimeh`	Upper 32 bits of `stime`, RV32I only.
`0xD82`	SRO	`sinstreth`	Upper 32 bits of `sinstret`, RV32I only.

Table 2.3: Currently allocated RISC-V supervisor-level CSR addresses.

Number	Privilege	Name	Description
Hypervisor Trap Setup			
`0x200`	HRW	`hstatus`	Hypervisor status register.
`0x202`	HRW	`hedeleg`	Hypervisor exception delegation register.
`0x203`	HRW	`hideleg`	Hypervisor interrupt delegation register.
`0x204`	HRW	`hie`	Hypervisor interrupt-enable register.
`0x205`	HRW	`htvec`	Hypervisor trap handler base address.
Hypervisor Trap Handling			
`0x240`	HRW	`hscratch`	Scratch register for hypervisor trap handlers.
`0x241`	HRW	`hepc`	Hypervisor exception program counter.
`0x242`	HRW	`hcause`	Hypervisor trap cause.
`0x243`	HRW	`hbadaddr`	Hypervisor bad address.
Hypervisor Protection and Translation			
`0x28X`	TBD	TBD	TBD.
Hypervisor Counter/Timers			
`0xE00`	HRO	`hcycle`	Hypervisor cycle counter.
`0xE01`	HRO	`htime`	Hypervisor wall-clock time.
`0xE02`	HRO	`hinstret`	Hypervisor instructions-retired counter.
`0xE80`	HRO	`hcycleh`	Upper 32 bits of `hcycle`, RV32I only.
`0xE81`	HRO	`htimeh`	Upper 32 bits of `htime`, RV32I only.
`0xE82`	HRO	`hinstreth`	Upper 32 bits of `hinstret`, RV32I only.

Table 2.4: Currently allocated RISC-V hypervisor-level CSR addresses.

Number	Privilege	Name	Description
Machine Information Registers			
`0xF10`	MRO	`misa`	ISA and extensions supported.
`0xF11`	MRO	`mvendorid`	Vendor ID.
`0xF12`	MRO	`marchid`	Architecture ID.
`0xF13`	MRO	`mimpid`	Implementation ID.
`0xF14`	MRO	`mhartid`	Hardware thread ID.
Machine Trap Setup			
`0x300`	MRW	`mstatus`	Machine status register.
`0x302`	MRW	`medeleg`	Machine exception delegation register.
`0x303`	MRW	`mideleg`	Machine interrupt delegation register.
`0x304`	MRW	`mie`	Machine interrupt-enable register.
`0x305`	MRW	`mtvec`	Machine trap-handler base address.
Machine Trap Handling			
`0x340`	MRW	`mscratch`	Scratch register for machine trap handlers.
`0x341`	MRW	`mepc`	Machine exception program counter.
`0x342`	MRW	`mcause`	Machine trap cause.
`0x343`	MRW	`mbadaddr`	Machine bad address.
`0x344`	MRW	`mip`	Machine interrupt pending.
Machine Protection and Translation			
`0x380`	MRW	`mbase`	Base register.
`0x381`	MRW	`mbound`	Bound register.
`0x382`	MRW	`mibase`	Instruction base register.
`0x383`	MRW	`mibound`	Instruction bound register.
`0x384`	MRW	`mdbase`	Data base register.
`0x385`	MRW	`mdbound`	Data bound register.
Machine Timers and Counters			
`0xF00`	MRO	`mcycle`	Machine cycle counter.
`0xF01`	MRO	`mtime`	Machine wall-clock time.
`0xF02`	MRO	`minstret`	Machine instructions-retired counter.
`0xF80`	MRO	`mcycleh`	Upper 32 bits of `mcycle`, RV32I only.
`0xF81`	MRO	`mtimeh`	Upper 32 bits of `mtime`, RV32I only.
`0xF82`	MRO	`minstreth`	Upper 32 bits of `minstret`, RV32I only.
Machine Counter Setup			
`0x310`	MRW	`mucounteren`	User-mode counter enable.
`0x311`	MRW	`mscounteren`	Supervisor-mode counter enable.
`0x312`	MRW	`mhcounteren`	Hypervisor-mode counter enable.

Table 2.5: Currently allocated RISC-V machine-level CSR addresses.

Number	Privilege	Name	Description
		Machine Counter-Delta Registers	
`0x700`	MRW	`mucycle_delta`	`cycle` counter delta.
`0x701`	MRW	`mutime_delta`	`time` counter delta.
`0x702`	MRW	`muinstret_delta`	`instret` counter delta.
`0x704`	MRW	`mscycle_delta`	`scycle` counter delta.
`0x705`	MRW	`mstime_delta`	`stime` counter delta.
`0x706`	MRW	`msinstret_delta`	`sinstret` counter delta.
`0x708`	MRW	`mhcycle_delta`	`hcycle` counter delta.
`0x709`	MRW	`mhtime_delta`	`htime` counter delta.
`0x70A`	MRW	`mhinstret_delta`	`hinstret` counter delta.
`0x780`	MRW	`mucycle_deltah`	Upper 32 bits of `cycle` counter delta, RV32I only.
`0x781`	MRW	`mutime_deltah`	Upper 32 bits of `time` counter delta, RV32I only.
`0x782`	MRW	`muinstret_deltah`	Upper 32 bits of `instret` counter delta, RV32I only.
`0x784`	MRW	`mscycle_deltah`	Upper 32 bits of `scycle` counter delta, RV32I only.
`0x785`	MRW	`mstime_deltah`	Upper 32 bits of `stime` counter delta, RV32I only.
`0x786`	MRW	`msinstret_deltah`	Upper 32 bits of `sinstret` counter delta, RV32I only.
`0x788`	MRW	`mhcycle_deltah`	Upper 32 bits of `hcycle` counter delta, RV32I only.
`0x789`	MRW	`mhtime_deltah`	Upper 32 bits of `htime` counter delta, RV32I only.
`0x78A`	MRW	`mhinstret_deltah`	Upper 32 bits of `hinstret` counter delta, RV32I only.

Table 2.6: Currently allocated RISC-V machine-level CSR addresses.

2.3 CSR Field Specifications

The following definitions and abbreviations are used in specifying the behavior of fields within the CSRs.

Reserved Writes Ignored, Reads Ignore Values (WIRI)

Some read-only and read/write registers have read-only fields reserved for future use. These reserved read-only fields should be ignored on a read. Writes to these fields have no effect, unless the whole CSR is read-only, in which case writes might raise an illegal instruction exception. These fields are labeled **WIRI** in the register descriptions.

Reserved Writes Preserve Values, Reads Ignore Values (WPRI)

Some whole read/write fields are reserved for future use. Software should ignore the values read from these fields, and should preserve the values held in these fields when writing values to other fields of the same register. These fields are labeled **WPRI** in the register descriptions.

To simplify the software model, any backward-compatible future definition of previously reserved fields within a CSR must cope with the possibility that a non-atomic read/modify/write sequence is used to update other fields in the CSR. Alternatively, the original CSR definition must specify that subfields can only be updated atomically, which may require a two-instruction clear bit/set bit sequence in general that can be problematic if intermediate values are not legal.

Write/Read Only Legal Values (WLRL)

Some read/write CSR fields specify behavior for only a subset of possible bit encodings, with other bit encodings reserved. Software should not write anything other than legal values to such a field, and should not assume a read will return a legal value unless the last write was of a legal value, or the register has not been written since another operation (e.g., reset) set the register to a legal value. These fields are labeled **WLRL** in the register descriptions.

Hardware implementations need only implement enough state bits to differentiate between the supported values, but must always return the complete specified bit-encoding of any supported value when read.

Implementations are permitted but not required to raise an illegal instruction exception if an instruction attempts to write a non-supported value to a CSR field. Hardware implementations can return arbitrary bit patterns on the read of a CSR field when the last write was of an illegal value, but the value returned should deterministically depend on the previous written value.

Write Any Values, Reads Legal Values (WARL)

Some read/write CSR fields are only defined for a subset of bit encodings, but allow any value to be written while guaranteeing to return a legal value whenever read. Assuming that writing the CSR has no other side effects, the range of supported values can be determined by attempting to write a desired setting then reading to see if the value was retained. These fields are labeled **WARL** in the register descriptions.

Implementations will not raise an exception on writes of unsupported values to an **WARL** field. Implementations must always deterministically return the same legal value after a given illegal value is written.

Chapter 3

Machine-Level ISA

This chapter describes the machine-level operations available in machine-mode (M-mode), which is the highest privilege mode in a RISC-V system. M-mode is the only mandatory privilege mode in a RISC-V hardware implementation. M-mode is used for low-level access to a hardware platform and is the first mode entered at reset. M-mode can also be used to implement features that are too difficult or expensive to implement in hardware directly. The RISC-V machine-level ISA contains a common core that is extended depending on which other privilege levels are supported and other details of the hardware implementation.

3.1 Machine-Level CSRs

In addition to the machine-level CSRs described in this section, M-mode code can access all CSRs at lower privilege levels.

3.1.1 Machine ISA Register `misa`

The `misa` register is an XLEN-bit read-only register reporting the ISA supported by the hart. This register must be readable in any implementation, but a value of zero can be returned to indicate the `misa` register has not been implemented, requiring that CPU capabilities be determined through a separate non-standard mechanism.

XLEN-1 XLEN-2	XLEN-3 26	25 0
Base	**WIRI**	Extensions
2	XLEN-28	26

Figure 3.1: Machine ISA register (`misa`).

The Base field encodes the native base integer ISA width as shown in Table 3.1. For implementations that support multiple base ISA widths, the Base field always describes the widest supported ISA variant as this is the ISA mode entered in machine-mode at reset.

The base can be quickly ascertained using branches on the sign of the returned `misa` *value, and*

Value	Description
1	32
2	64
3	128

Table 3.1: Encoding of Base field in `misa`

possibly a shift left by one and a second branch on the sign. These checks can be written in assembly code without knowing the register width (XLEN) of the machine. The base width is given by $XLEN = 2^{Base+4}$.

The Extensions field encodes the presence of the standard extensions, with a single bit per letter of the alphabet (bit 0 encodes presence of extension “A” , bit 1 encodes presence of extension “B”, through to bit 25 which encodes “Z”). The “I” bit will be set for RV32I, RV64I, RV128I base ISAs, and the “E” bit will be set for RV32E.

The “G” bit is used as an escape to allow expansion to a larger space of standard extension names.

G is used to indicate the combination IMAFD, so is redundant in the `misa` *register, hence we reserve the bit to indicate that additional standard extensions are present.*

The “U”,“S”, and “H” bits will be set if there is support for user, supervisor, and hypervisor privilege modes respectively.

The “X” bit will be set if there are any non-standard extensions.

The `misa` *register exposes a rudimentary catalog of CPU features to machine-mode code. More extensive information can be obtained in machine mode by probing other machine registers, and examining other ROM storage in the system as part of the boot process.*

We require that lower privilege levels execute environment calls instead of reading CPU registers to determine features available at each privilege level. This enables virtualization layers to alter the ISA observed at any level, and supports a much richer command interface without burdening hardware designs.

Bit	Character	Description
0	A	Atomic extension
1	B	*Tentatively reserved for Bit operations extension*
2	C	Compressed extension
3	D	Double-precision floating-point extension
4	E	RV32E base ISA
5	F	Single-precision floating-point extension
6	G	Additional standard extensions present
7	H	Hypervisor mode implemented
8	I	RV32I/64I/128I base ISA
9	J	*Reserved*
10	K	*Reserved*
11	L	*Tentatively reserved for Decimal Floating-Point extension*
12	M	Integer Multiply/Divide extension
13	N	User-level interrupts supported
14	O	*Reserved*
15	P	*Tentatively reserved for Packed-SIMD extension*
16	Q	Quad-precision floating-point extension
17	R	*Reserved*
18	S	Supervisor mode implemented
19	T	*Tentatively reserved for Transactional Memory extension*
20	U	User mode implemented
21	V	*Tentatively reserved for Vector extension*
22	W	*Reserved*
23	X	Non-standard extensions present
24	Y	*Reserved*
25	Z	*Reserved*

Table 3.2: Encoding of Base field in `misa`. All bits that are reserved for future use must return zero when read.

3.1.2 Machine Vendor ID Register `mvendorid`

The `mvendorid` CSR is an XLEN-bit read-only register encoding the manufacturer of the part. This register must be readable in any implementation, but a value of 0 can be returned to indicate the field is not implemented or that this is a non-commercial implementation.

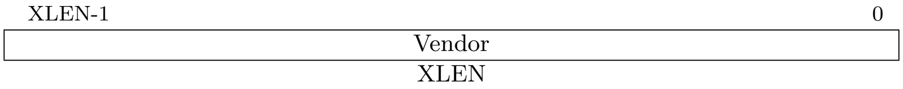

Figure 3.2: Vendor ID register (`mvendorid`).

Non-zero vendor IDs will be allocated by the RISC-V Foundation to commercial vendors of RISC-V chips.

3.1.3 Machine Architecture ID Register `marchid`

The `marchid` CSR is an XLEN-bit read-only register encoding the base microarchitecture of the hart. This register must be readable in any implementation, but a value of 0 can be returned to indicate the field is not implemented. The combination of `mvendorid` and `marchid` should uniquely identify the type of hart microarchitecture that is implemented.

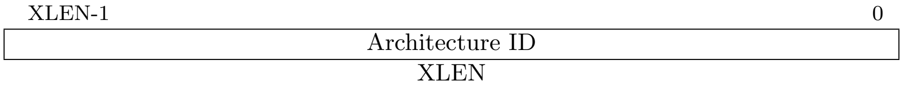

Figure 3.3: Machine Architecture ID register (`marchid`).

Open-source project architecture IDs are allocated globally by the RISC-V Foundation, and have non-zero architecture IDs with a zero most-significant-bit (MSB). Commercial architecture IDs are allocated by each commercial vendor independently, but must have the MSB set and cannot contain zero in the remaining XLEN-1 bits.

The intent is for the architecture ID to represent the microarchitecture associated with the repo around which development occurs rather than a particular organization. Commercial fabrications of open-source designs should (and might be required by the license to) retain the original architecture ID. This will aid in reducing fragmentation and tool support costs, as well as provide attribution. Open-source architecture IDs should be administered by the Foundation and should only be allocated to released, functioning open-source projects. Commercial architecture IDs can be managed independently by any registered vendor but are required to have IDs disjoint from the open-source architecture IDs (MSB set) to prevent collisions if a vendor wishes to use both closed-source and open-source microarchitectures.

The convention adopted within the following Implementation field can be used to segregate branches of the same architecture design, including by organization. The `misa` *register also helps distinguish different variants of a design, as does the configuration string if present.*

3.1.4 Machine Implementation ID Register `mimpid`

The `mimpid` CSR provides a unique encoding of the version of the processor implementation. This register must be readable in any implementation, but a value of 0 can be returned to indicate that the field is not implemented. The Implementation value should reflect the design of the RISC-V processor itself and not any surrounding system.

XLEN-1	0
Implementation	
XLEN	

Figure 3.4: Machine Implementation ID register (`mimpid`).

The format of this field is left to the provider of the architecture source code, but will be often be printed by standard tools as a hexadecimal string without any leading or trailing zeros, so the Implementation value should be left-justified (i.e., filled in from most-significant nibble down) with subfields aligned on nibble boundaries to ease human readability.

3.1.5 Hart ID Register `mhartid`

The `mhartid` register is an XLEN-bit read-only register containing the integer ID of the hardware thread running the code. This register must be readable in any implementation. Hart IDs might not necessarily be numbered contiguously in a multiprocessor system, but at least one hart must have a hart ID of zero.

XLEN-1	0
Hart ID	
XLEN	

Figure 3.5: Hart ID register (`mhartid`).

In certain cases, we must ensure exactly one hart runs some code (e.g., at reset), and so require one hart to have a known hart ID of zero.

For efficiency, system implementers should aim to reduce the magnitude of the largest hart ID used in a system.

3.1.6 Machine Status Register (`mstatus`)

The `mstatus` register is an XLEN-bit read/write register formatted as shown in Figure 3.6. The `mstatus` register keeps track of and controls the hart's current operating state. Restricted views of the `mstatus` register appear as the `hstatus` and `sstatus` registers in the H and S privilege-level ISAs respectively.

3.1.7 Privilege and Global Interrupt-Enable Stack in `mstatus` register

Interrupt-enable bits, MIE, HIE, SIE, and UIE, are provided for each privilege mode. These bits are primarily used to guarantee atomicity with respect to interrupt handlers at the current privilege level. When a hart is executing in privilege mode x, interrupts are enabled when xIE=1. Interrupts for lower privilege modes are always disabled, whereas interrupts for higher privilege modes are always enabled. Higher-privilege-level code can use separate per-interrupt enable bits to disable selected interrupts before ceding control to a lower privilege level.

XLEN-1	XLEN-2 29	28 24	23 20	19	18	17	16 15
SD	**WPRI**	VM[4:0] (**WARL**)	**WPRI**	MXR	PUM	MPRV	XS[1:0]
1	XLEN-30	5	4	1	1	1	2

14 13	12 11	10 9	8	7	6	5	4	3	2	1	0
FS[1:0]	MPP[1:0]	HPP[1:0]	SPP	MPIE	HPIE	SPIE	UPIE	MIE	HIE	SIE	UIE
2	2	2	1	1	1	1	1	1	1	1	1

Figure 3.6: Machine-mode status register (`mstatus`).

The x*IE bits are located in the low-order bits of* `mstatus`*, allowing them to be atomically set or cleared with a single CSR instruction.*

To support nested traps, each privilege mode x has a two-level stack of interrupt-enable bits and privilege modes. xPIE holds the value of the interrupt-enable bit active prior to the trap, and xPP holds the previous privilege mode. The xPP fields can only hold privilege modes up to x, so MPP and HPP are two bits wide, SPP is one bit wide, and UPP is implicitly zero. When a trap is taken from privilege mode y into privilege mode x, xPIE is set to the value of yIE; xIE is set to 0; and xPP is set to y.

For lower privilege modes, any trap (synchronous or asynchronous) is usually taken at a higher privilege mode with interrupts disabled. The higher-level trap handler will either service the trap and return using the stacked information, or, if not returning immediately to the interrupted context, will save the privilege stack before re-enabling interrupts, so only one entry per stack is required.

The MRET, HRET, SRET, or URET instructions are used to return from traps in M-mode, H-mode, S-mode, or U-mode respectively. When executing an xRET instruction, supposing xPP holds the value y, yIE is set to xPIE; the privilege mode is changed to y; xPIE is set to 1; and xPP is set to U (or M if user-mode is not supported).

When the stack is popped, the lowest-supported privilege mode with interrupts enabled is added to the bottom of stack to help catch errors that cause invalid entries to be popped off the stack.

xPP fields are **WLRL** fields that need only be able to store supported privilege modes.

If the machine provides only U and M modes, then only a single hardware storage bit is required to represent either 00 or 11 in MPP. If the machine provides only M mode, then MPP is hardwired to 11.

User-level interrupts are an optional extension and have been allocated the ISA extension letter N. If user-level interrupts are omitted, the UIE and UPIE bits are hardwired to zero. For all other supported privilege modes x, the xIE, xPIE, and xPP fields are required to be implemented.

User-level interrupts are primarily intended to support secure embedded systems with only M-mode and U-mode present.

3.1.8 Virtualization Management Field in `mstatus` Register

The virtualization management field VM[4:0] indicates the currently active scheme for virtualization, including virtual memory translation and protection. Table 3.3 shows the currently defined virtualization schemes. Only the Mbare mode is mandatory for a RISC-V hardware implementation. The Mbare, Mbb, and Mbbid schemes are described in Sections 3.7–3.8, while the page-based virtual memory schemes are described in later chapters.

Each setting of the VM field defines operation at all supported privilege levels, and the behavior of some VM settings might differ depending on the privilege levels supported in hardware.

Value	Abbreviation	Modes Required	Description
0	Mbare	M	No translation or protection.
1	Mbb	M, U	Single base-and-bound.
2	Mbbid	M, U	Separate instruction and data base-and-bound.
3–7	*Reserved*		
8	Sv32	M, S, U	Page-based 32-bit virtual addressing.
9	Sv39	M, S, U	Page-based 39-bit virtual addressing.
10	Sv48	M, S, U	Page-based 48-bit virtual addressing.
11	Sv57	M, S, U	Reserved for page-based 57-bit virtual addressing.
12	Sv64	M, S, U	Reserved for page-based 64-bit virtual addressing.
13–31	*Reserved*		

Table 3.3: Encoding of virtualization management field VM[4:0].

Mbare corresponds to no memory management or translation, and so all effective addresses regardless of privilege mode are treated as machine physical addresses. Mbare is the mode entered at reset.

Mbb is a base-and-bounds architectures for systems with at least two privilege levels (U and M). Mbb is suited for systems that require low-overhead translation and protection for user-mode code, and that do not require demand-paged virtual memory (swapping is supported). A variant Mbbid provides separate address and data segments to allow an execute-only code segment to be shared between processes.

Sv32 is a page-based virtual-memory architecture for RV32 systems providing a 32-bit virtual address space designed to support modern supervisor-level operating systems, including Unix-based systems.

Sv39 and Sv48 are page-based virtual-memory architectures for RV64 systems providing a 39-bit or 48-bit virtual address space respectively to support modern supervisor-level operating systems, including Unix-based systems.

Sv32, Sv39, and Sv48 require implementations to support M, S, and U privilege levels. If H-mode is also present, additional operations are defined for hypervisor-level code to support multiple supervisor-level virtual machines. Hypervisor-mode support for virtual machines has not yet been defined.

The existing Sv39 and Sv48 schemes can be readily extended to Sv57 and Sv64 virtual address *widths. Sv52, Sv60, Sv68, and Sv76 virtual address space widths are tentatively planned for RV128 systems, where virtual address widths under 68 bits are intended for applications requiring 128-bit integer arithmetic but not larger address spaces.*

Wider virtual address widths incur microarchitectural costs for wider internal registers as well as longer page-table searches on address-translation cache misses, so we support a range of virtual address widths where each wider width adds one more level to the in-memory page table. A single hardware page-table walker design can easily support multiple virtual address widths, but requires internal hardware registers to support the widest width.

Our current definition of the virtualization management schemes only supports the same base architecture at every privilege level. Variants of the virtualization schemes can be defined to support narrow widths at lower-privilege levels, e.g., to run RV32 code on an RV64 system.

VM is a **WARL** field, so whether a VM setting is supported by an implementation can be determined by writing the value to VM, then reading the value back from VM to see if the same value was returned.

3.1.9 Memory Privilege in `mstatus` Register

The MPRV bit modifies the privilege level at which loads and stores execute. When MPRV=0, translation and protection behave as normal. When MPRV=1, data memory addresses are translated and protected as though the current privilege mode were set to MPP. Instruction address-translation and protection are unaffected.

The MXR (Make eXecutable Readable) bit modifies the privilege with which loads access virtual memory. When MXR=0, only loads from pages marked readable (R=1 in Figure 4.19) will succeed. When MXR=1, loads from pages marked either readable or executable (R=1 or X=1) will succeed.

The MPRV and MXR mechanisms were conceived to improve the efficiency of M-mode routines that emulate missing hardware features, e.g., misaligned loads and stores. MPRV obviates the need to perform address translation in software. MXR allows instruction words to be loaded from pages marked execute-only.

For simplicity, MPRV and MXR are in effect regardless of privilege mode, but in normal use will only be enabled for short sequences in machine mode.

The PUM (Protect User Memory) bit modifies the privilege with which S-mode loads, stores, and instruction fetches access virtual memory. When PUM=0, translation and protection behave as normal. When PUM=1, S-mode memory accesses to pages that are accessible by U-mode (U=1 in Figure 4.19) will fault. PUM has no effect when page-based virtual memory is not in effect. Note that, while PUM is ordinarily ignored when not executing in S-mode, it *is* in effect when MPRV=1 and MPP=S.

3.1.10 Extension Context Status in `mstatus` Register

Supporting substantial extensions is one of the primary goals of RISC-V, and hence we define a standard interface to allow unchanged privileged-mode code, particularly a supervisor-level OS, to support arbitrary user-mode state extensions.

To date, there are no standard extensions that define additional state beyond the floating-point CSR and data registers.

The FS[1:0] read/write field and the XS[1:0] read-only field are used to reduce the cost of context save and restore by setting and tracking the current state of the floating-point unit and any other user-mode extensions respectively. The FS field encodes the status of the floating-point unit, including the CSR `fcsr` and floating-point data registers `f0`–`f31`, while the XS field encodes the status of any additional user-mode extensions and associated state. These fields can be checked by a context switch routine to quickly determine whether a state save or restore is required. If a save or restore is required, additional instructions and CSRs are typically required to effect and optimize the process.

The design anticipates that most context switches will not need to save/restore state in either or both of the floating-point unit or other extensions, so provides a fast check via the SD bit.

The FS and XS fields use the same status encoding as shown in Table 3.4, with the four possible status values being Off, Initial, Clean, and Dirty.

Status	FS Meaning	XS Meaning
0	Off	All off
1	Initial	None dirty or clean, some on
2	Clean	None dirty, some clean
3	Dirty	Some dirty

Table 3.4: Encoding of FS[1:0] and XS[1:0] status fields.

In systems that do not implement S-mode and do not have a floating-point unit, the FS field is hardwired to zero.

In systems without additional user extensions requiring new state, the XS field is hardwired to zero. Every additional extension with state has a local status register encoding the equivalent of the XS states. If there is only a single additional extension, its status can be directly mirrored in the XS field. If there is more than one additional extension, the XS field represents a summary of all extensions' status as shown in Table 3.4.

The XS field effectively reports the maximum status value across all user-extension status fields, though individual extensions can use a different encoding than XS.

The SD bit is a read-only bit that summarizes whether either the FS field or XS field signals the presence of some dirty state that will require saving extended user context to memory. If both XS and FS are hardwired to zero, then SD is also always zero.

When an extension's status is set to Off, any instruction that attempts to read or write the corresponding state will cause an exception. When the status is Initial, the corresponding state should have an initial constant value. When the status is Clean, the corresponding state is potentially different from the initial value, but matches the last value stored on a context swap. When the status is Dirty, the corresponding state has potentially been modified since the last context save.

During a context save, the responsible privileged code need only write out the corresponding state if its status is Dirty, and can then reset the extension's status to Clean. During a context restore, the context need only be loaded from memory if the status is Clean (it should never be Dirty at restore). If the status is Initial, the context must be set to an initial constant value on context restore to avoid a security hole, but this can be done without accessing memory. For example, the floating-point registers can all be initialized to the immediate value 0.

The FS and XS fields are read by the privileged code before saving the context. The FS field is set directly by privileged code when resuming a user context, while the XS field is set indirectly by writing to the status register of the individual extensions. The status fields will also be updated during execution of instructions, regardless of privilege mode.

Extensions to the user-mode ISA often include additional user-mode state, and this state can be considerably larger than the base integer registers. The extensions might only be used for some applications, or might only be needed for short phases within a single application. To improve performance, the user-mode extension can define additional instructions to allow user-mode software to return the unit to an initial state or even to turn off the unit.

For example, a coprocessor might require to be configured before use and can be "unconfigured" after use. The unconfigured state would be represented as the Initial state for context save. If the same application remains running between the unconfigure and the next configure (which would set status to Dirty), there is no need to actually reinitialize the state at the unconfigure instruction, as all state is local to the user process, i.e., the Initial state may only cause the coprocessor state to be initialized to a constant value at context restore, not at every unconfigure.

Executing a user-mode instruction to disable a unit and place it into the Off state will cause an illegal instruction exception to be raised if any subsequent instruction tries to use the unit before it is turned back on. A user-mode instruction to turn a unit on must also ensure the unit's state is properly initialized, as the unit might have been used by another context meantime.

Table 3.5 shows all the possible state transitions for the FS or XS status bits. Note that the standard floating-point extensions do not support user-mode unconfigure or disable/enable instructions.

Standard privileged instructions to initialize, save, and restore extension state are provided to insulate privileged code from details of the added extension state by treating the state as an opaque object.

Many coprocessor extensions are only used in limited contexts that allows software to safely unconfigure or even disable units when done. This reduces the context-switch overhead of large stateful coprocessors.

We separate out floating-point state from other extension state, as when a floating-point unit is present the floating-point registers are part of the standard calling convention, and so user-mode software cannot know when it is safe to disable the floating-point unit.

The XS field provides a summary of all added extension state, but additional microarchitectural bits might be maintained in the extension to further reduce context save and restore overhead.

The SD bit is read-only and is set when either the FS or XS bits encode a Dirty state (i.e., SD=((FS==11) OR (XS==11))). This allows privileged code to quickly determine when no additional context save is required beyond the integer register set and PC.

The floating-point unit state is always initialized, saved, and restored using standard instructions (F, D, and/or Q), and privileged code must be aware of FLEN to determine the appropriate space

Current State	Off	Initial	Clean	Dirty
Action				
At context save in privileged code				
Save state?	No	No	No	Yes
Next state	Off	Initial	Clean	Clean
At context restore in privileged code				
Restore state?	No	Yes, to initial	Yes, from memory	N/A
Next state	Off	Initial	Clean	N/A
Execute instruction to read state				
Action?	Exception	Execute	Execute	Execute
Next state	Off	Initial	Clean	Dirty
Execute instruction to modify state, including configuration				
Action?	Exception	Execute	Execute	Execute
Next state	Off	Dirty	Dirty	Dirty
Execute instruction to unconfigure unit				
Action?	Exception	Execute	Execute	Execute
Next state	Off	Initial	Initial	Initial
Execute instruction to disable unit				
Action?	Execute	Execute	Execute	Execute
Next state	Off	Off	Off	Off
Execute instruction to enable unit				
Action?	Execute	Execute	Execute	Execute
Next state	Initial	Initial	Initial	Initial

Table 3.5: FS and XS state transitions.

to reserve for each `f` register.

In a supervisor-level OS, any additional user-mode state should be initialized, saved, and restored using SBI calls that treats the additional context as an opaque object of a fixed maximum size. The implementation of the SBI initialize, save, and restore calls might require additional implementation-dependent privileged instructions to initialize, save, and restore microarchitectural state inside a coprocessor.

All privileged modes share a single copy of the FS and XS bits. In a system with more than one privileged mode, supervisor mode would normally use the FS and XS bits directly to record the status with respect to the supervisor-level saved context. Other more-privileged active modes must be more conservative in saving and restoring the extension state in their corresponding version of the context, but can rely on the Off state to avoid save and restore, and the Initial state to avoid saving the state.

In any reasonable use case, the number of context switches between user and supervisor level should far outweigh the number of context switches to other privilege levels. Note that coprocessors should not require their context to be saved and restored to service asynchronous interrupts, unless the interrupt results in a user-level context swap.

3.1.11 Machine Trap-Vector Base-Address Register (`mtvec`)

The `mtvec` register is an XLEN-bit read/write register that holds the base address of the M-mode trap vector.

XLEN-1 2	1 0
Trap-Vector Base Address (**WARL**)	0
XLEN-2	2

Figure 3.7: Machine trap-vector base-address register (`mtvec`).

The `mtvec` register must always be implemented, but can contain a hardwired read-only value. If `mtvec` is writable, the set of values the register may hold can vary by implementation. The value in the `mtvec` register must always be aligned on a 4-byte boundary (the low two bits are always zero). The value returned by reading a variable `mtvec` register should always match the value used to generate the handler PC address when handling traps.

We allow for considerable flexibility in implementation of the trap vector base address. On the one hand, we do not wish to burden low-end implementations with a large number of state bits, but on the other hand, we wish to allow flexibility for larger systems.

By default, all traps into machine mode cause the `pc` to be set to the value in `mtvec`. Additional trap vector entry points can be defined by implementations to allow more rapid identification and service of certain trap causes.

The location of the reset vector and non-maskable interrupt vector are implementation-defined.

Reset, NMI vectors, and other interrupt vector default locations are given in a platform specification.

3.1.12 Machine Trap Delegation Registers (`medeleg` and `mideleg`)

By default, all traps at any privilege level are handled in machine mode, though a machine-mode handler can redirect traps back to the appropriate level with the MRET instruction (Section 3.2.1). To increase performance, implementations can provide individual read/write bits within `medeleg` and `mideleg` to indicate that certain exceptions and interrupts should be processed directly by a lower privilege level. The machine exception delegation register (`medeleg`) and machine interrupt delegation register (`mideleg`) are XLEN-bit read/write registers.

In systems with all four privilege modes (M/H/S/U), a set bit in `medeleg` or `mideleg` will delegate any corresponding trap in U-mode, S-mode, or H-mode to the H-mode trap handler. H-mode may in turn set corresponding bits in the `hedeleg` and `hideleg` registers to delegate traps that occur in S-mode or U-mode to the S-mode trap handler. If U-mode traps are supported, S-mode may in turn set corresponding bits in the `sedeleg` and `sideleg` registers to delegate traps that occur in U-mode to the U-mode trap handler.

In systems with three privilege modes (M/S/U), setting a bit in `medeleg` or `mideleg` will delegate the corresponding trap in S-mode or U-mode to the S-mode trap handler. If U-mode traps are supported, S-mode may in turn set corresponding bits in the `sedeleg` and `sideleg` registers to delegate traps that occur in U-mode to the U-mode trap handler.

In systems with two privilege modes (M/U) and support for U-mode traps, setting a bit in `medeleg` or `mideleg` will delegate the corresponding trap in U-mode to the U-mode trap handler.

If systems with only M-mode, or with both M-mode and U-mode but without U-mode trap support, the `medeleg` and `mideleg` registers should be hardwired to zero.

When a trap is delegated to a less-privileged mode x, the x`cause` register is written with the trap cause; the x`epc` register is written with the virtual address of the instruction that took the trap; the xPP field of `mstatus` is written with the active privilege mode at the time of the trap; the xPIE field of `mstatus` is written with the value of the active interrupt-enable bit at the time of the trap; and the xIE field of `mstatus` is cleared. The `mcause` and `mepc` registers and the MPP and MPIE fields of `mstatus` are not written.

An implementation shall not hardwire any delegation bits to one, i.e., any trap that can be delegated must support not being delegated. An implementation can choose to subset the delegatable traps, with the supported delegatable bits found by writing one to every bit location, then reading back the value in `medeleg` or `mideleg` to see which bit positions hold a one.

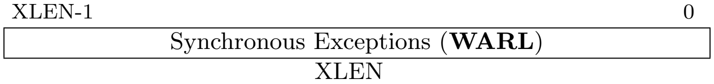

Figure 3.8: Machine Exception Delegation Register `medeleg`.

`medeleg` has a bit position allocated for every synchronous exception shown in Table 3.6, with the index of the bit position equal to the value returned in the `mcause` register (i.e., setting bit 8 allows user-mode environment calls to be delegated to a lower-privilege trap handler).

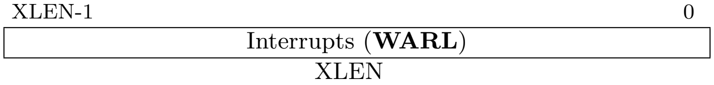

Figure 3.9: Machine Exception Delegation Register `mideleg`.

`mideleg` holds trap delegation bits for individual interrupts, with the layout of bits matching those in the `mip` register (i.e., STIP interrupt delegation control is located in bit 5).

3.1.13 Machine Interrupt Registers (`mip` and `mie`)

The `mip` register is an XLEN-bit read/write register containing information on pending interrupts, while `mie` is the corresponding XLEN-bit read/write register containing interrupt enable bits. Only the bits corresponding to lower-privilege software interrupts (USIP, SSIP, HSIP) and timer interrupts (UTIP, STIP and HTIP) in `mip` are writable through this CSR address; the remaining bits are read-only.

Restricted views of the `mip` and `mie` registers appear as the `hip`/`hie`, `sip`/`sie`, and `uip`/`uie` registers in H-mode, S-mode, and U-mode respectively. If an interrupt is delegated to privilege mode x by setting a bit in the `mideleg` register, it becomes visible in the *x*`ip` register and is maskable using the *x*`ie` register. Otherwise, the corresponding bits in *x*`ip` and *x*`ie` appear to be hardwired to zero.

XLEN-1 12	11	10	9	8	7	6	5	4	3	2	1	0
WIRI	MEIP	HEIP	SEIP	UEIP	MTIP	HTIP	STIP	UTIP	MSIP	HSIP	SSIP	USIP
XLEN-12	1	1	1	1	1	1	1	1	1	1	1	1

Figure 3.10: Machine interrupt-pending register (`mip`).

XLEN-1 12	11	10	9	8	7	6	5	4	3	2	1	0
WPRI	MEIE	HEIE	SEIE	UEIE	MTIE	HTIE	STIE	UTIE	MSIE	HSIE	SSIE	USIE
XLEN-12	1	1	1	1	1	1	1	1	1	1	1	1

Figure 3.11: Machine interrupt-enable register (`mie`).

The MTIP, HTIP, STIP, UTIP bits correspond to timer interrupt-pending bits for machine, hypervisor, supervisor, and user timer interrupts, respectively. The MTIP bit is read-only and is cleared by writing to the memory-mapped machine-mode timer compare register. The UTIP, STIP and HTIP bits may be written by M-mode software to deliver timer interrupts to lower privilege levels. User, supervisor and hypervisor software may clear the UTIP, STIP and HTIP bits with calls to the AEE, SEE, or HEE, respectively.

There is a separate timer interrupt-enable bit, named MTIE, HTIE, STIE, and UTIE for M-mode, H-mode, S-mode, and U-mode timer interrupts respectively.

Each lower privilege level has a separate software interrupt-pending bit (HSIP, SSIP, USIP), which can be both read and written by CSR accesses from code running on the local hart at the associated or any higher privilege level. The machine-level MSIP bits are written by accesses to memory-mapped control registers, which are used by remote harts to provide machine-mode interprocessor interrupts. Interprocessor interrupts for lower privilege levels are implemented through ABI, SBI or HBI calls to the AEE, SEE or HEE respectively, which might ultimately result in a machine-mode write to the receiving hart's MSIP bit. A hart can write its own MSIP bit using the same memory-mapped control register.

We only allow a hart to directly write its own HSIP, SSIP, or USIP bits when running in appropriate mode, as other harts might be virtualized and possibly descheduled by higher privilege levels. We rely on ABI, SBI, and HBI calls to provide interprocessor interrupts for this reason. Machine-mode harts are not virtualized and can directly interrupt other harts by setting their MSIP bits, typically using uncached I/O writes to memory-mapped control registers depending on the platform specification.

The MEIP, HEIP, SEIP, UEIP bits correspond to external interrupt-pending bits for machine, hypervisor, supervisor, and user external interrupts, respectively. These bits are read-only and are set and cleared by a platform-specific interrupt controller, such as the standard platform-level interrupt controller specified in Chapter 7. There is a separate external interrupt-enable bit, named MEIE, HEIE, SEIE, and UEIE for M-mode, H-mode, S-mode, and U-mode external interrupts respectively.

The non-maskable interrupt is not made visible via the `mip` *register as its presence is implicitly known when executing the NMI trap handler.*

For all the various interrupt types (software, timer, and external), if a privilege level is not supported, the associated pending and interrupt-enable bits are hardwired to zero in the `mip` and `mie` registers respectively. Hence, these are all effectively **WARL** fields.

Implementations can add additional platform-specific machine-level interrupt sources to the high bits of these registers, though the expectation is that most external interrupts will be routed through the platform interrupt controller and be delivered via MEIP.

An interrupt i will be taken if bit i is set in both `mip` and `mie`, and if interrupts are globally enabled. By default, M-mode interrupts are globally enabled if the hart's current privilege mode is less than M, or if the current privilege mode is M and the MIE bit in the `mstatus` register is set. If bit i in `mideleg` is set, however, interrupts are considered to be globally enabled if the hart's current privilege mode equals the delegated privilege mode (H, S, or U) and that mode's interrupt enable bit (HIE, SIE or UIE in `mstatus`) is set, or if the current privilege mode is less than the delegated privilege mode.

Multiple simultaneous interrupts and traps at the same privilege level are handled in the following decreasing priority order: external interrupts, software interrupts, timer interrupts, then finally any synchronous traps.

3.1.14 Machine Timer Registers (`mtime` and `mtimecmp`)

M-mode includes a timer facility provided by the high-resolution read-only real-time counter `mtime`. The hardware platform must provide a facility for determining the timebase of `mtime`, which must run at a constant frequency.

The `mtime` register has a 64-bit precision on all RV32, RV64, and RV128 systems. Platforms provide a 64-bit memory-mapped machine-mode timer compare register (`mtimecmp`), which causes a timer interrupt to be posted when the `mtime` register contains a value greater than or equal to the value in the `mtimecmp` register. The interrupt remains posted until it is cleared by writing the `mtimecmp` register. The interrupt will only be taken if interrupts are enabled and the MTIE bit is set in the `mie` register.

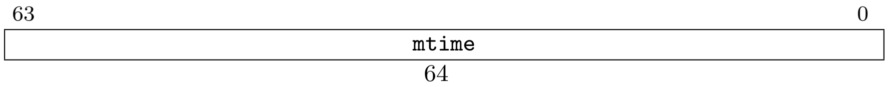

Figure 3.12: Machine time register.

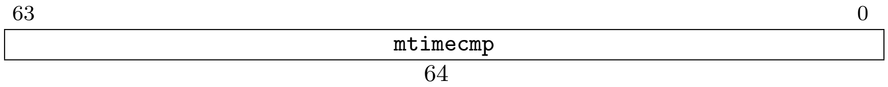

Figure 3.13: Machine time compare register (memory-mapped control register).

The timer facility is defined to use wall-clock time rather than a cycle counter to support modern processors that run with a highly variable clock frequency to save energy through dynamic voltage and frequency scaling.

Accurate real-time clocks (RTCs) are relatively expensive to provide (requiring a crystal or MEMS oscillator) and have to run even when the rest of system is powered down, and so there is usually only one in a system located in a different frequency/voltage domain from the processors. Hence, the RTC must be shared by all the harts in a system and accesses to the RTC will potentially incur the penalty of a voltage-level-shifter and clock-domain crossing. We assume the RTC will be exposed via memory-mapped registers at the platform level, and so one microarchitectural implementation strategy is to convert reads of the `mtime` *CSR into uncached reads of the platform RTC, reusing the existing memory-mapped path to the RTC. The same technique can be used at lower privilege levels to reduce the overhead of accessing* `htime`*,* `stime`*, and* `time`*.*

The timer compare register `mtimecmp` *is provided as a memory-mapped register rather than a CSR, as it must live in the same voltage/frequency domain as the real-time clock, and is accessed much less frequently than the timer value itself. Every hart has a unique memory-mapped* `mtimecmp` *register located next to the RTC to generate timer interrupts. Given an underlying RTC and timer compare register, machine-mode software can implement any number of virtual timers on a hart by multiplexing the next timer interrupt into the* `mtimecmp` *register.*

Simple fixed-frequency systems can use a single clock for both cycle counting and wall-clock time.

On RV32 only, reads of the `mtime` CSR return the low 32 bits of `mtime`, while reads of the `mtimeh` CSR return bits 63–32 of `mtime`.

On RV128 systems, the 64-bit value in `mtime` is sign-extended to 128-bits when read.

On RV128 systems, both signed and unsigned 64-bit values are held in a canonical form with bit 63 repeated in all higher bit positions. The timer is a 64-bit value even in RV128, and so the timer CSR read preserves the sign-extension invariant. Implementations may choose to implement fewer bits of the timer, provided software would be unlikely to experience timer wraparound (e.g., 100 years of wall clock time), and thereby avoid having to actually implement the sign-extension circuitry.

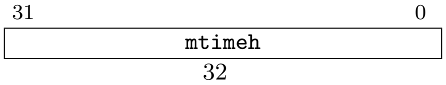

Figure 3.14: Upper 32 bits of machine time register, RV32 only.

In RV32, memory-mapped writes to `mtimecmp` modify only one 32-bit part of the register. The following code sequence sets a 64-bit `mtimecmp` value without spuriously generating a timer interrupt due to the intermediate value of the comparand:

```
# New comparand is in a1:a0.
li   t0, -1
sw t0, mtimecmp   # No smaller than old value.
sw a1, mtimecmp+4 # No smaller than new value.
sw a0, mtimecmp   # New value.
```

Figure 3.15: Sample code for setting the 64-bit time comparand in RV32 assuming the registers live in a strongly ordered I/O region.

3.1.15 Machine Performance Counters (mcycle, minstret)

M-mode includes a basic performance measurement facility. The `mcycle` CSR holds a count of the number of cycles the hart has executed since some arbitrary time in the past. The `minstret` CSR holds a count of the number of instructions the hart has retired since some arbitrary time in the past. The `mcycle` and `minstret` registers have 64-bit precision on all RV32, RV64, and RV128 systems.

63 … 0
mcycle
64

Figure 3.16: Machine cycle counter.

63 … 0
minstret
64

Figure 3.17: Machine instructions retired counter.

On RV32 only, reads of the `mcycle` and `minstret` CSRs return the low 32 bits, while reads of `mcycleh` and `minstreth` CSRs return bits 63–32 of `mcycle` and `minstret`, respectively.

On RV128 systems, the 64-bit values in `mcycle` and `minstret` are sign-extended to 128-bits when read.

31 … 0
mcycleh
32

Figure 3.18: Upper 32 bits of machine cycle counter, RV32 only.

31 … 0
minstreth
32

Figure 3.19: Upper 32 bits of machine instructions retired counter, RV32 only.

3.1.16 Machine Counter-Enable Registers (m[h|s|u]counteren)

XLEN-1 … 3	2	1	0
WIRI	IR	TM	CY
XLEN-3	1	1	1

Figure 3.20: Machine counter-enable registers (`mhcounteren`, `mscounteren`, `mucounteren`).

The machine counter-enable registers, `mhcounteren`, `mscounteren`, and `mucounteren`, control the availability of the hypervisor, supervisor, and user counters, respectively.

When the CY, TM, or IR bit in the `mhcounteren` register is clear, attempts to read the `hcycle`, `htime`, or `hinstret` register will cause an illegal instruction exception. When one of these bits is set, access to the corresponding register is permitted. The CY, TM, and IR bits in the `mscounteren` register analogously control access to the `scycle`, `stime`, and `sinstret` registers. The same bit positions in the `mucounteren` register analogously control access to the `cycle`, `time`, and `instret` registers.

Each counter-enable register must be implemented if the corresponding privilege mode is implemented. However, any of the bits may contain a hardwired value of zero, indicating reads to the corresponding counter will cause an exception. Hence, they are effectively **WARL** fields.

3.1.17 Machine Counter-Delta Registers

63 0
`mhcycle_delta` (**WARL**)
`mscycle_delta` (**WARL**)
`mucycle_delta` (**WARL**)
`mhtime_delta` (**WARL**)
`mstime_delta` (**WARL**)
`mutime_delta` (**WARL**)
`mhinstret_delta` (**WARL**)
`msinstret_delta` (**WARL**)
`muinstret_delta` (**WARL**)
64

Figure 3.21: Machine counter-delta registers.

The machine counter-delta registers give the difference between a less-privileged counter and the corresponding M-mode counter (e.g., `stime`−`mtime`=`mstime_delta`, and `instret`−`minstret`=`muinstret_delta`). Reading the less privileged counter gives the sum of the M-mode counter and the corresponding delta register.

Each counter-delta register must be implemented if the corresponding privilege mode is implemented. However, the counter may be hardwired to zero, indicating the M-mode counter and less-privileged counter have the same value.

On implementations that have hardwired delta registers to zero, M-mode software can emulate their functionality by maintaining the deltas in memory, clearing the corresponding bits in the counter-enable registers, and adding the deltas in the M-mode illegal instruction exception handler.

For RV32 only, the upper 32 bits of each delta register are aliased by another CSR, shown in Figure 3.22.

The combination of counter enable bits and counter-delta registers was designed to support common use cases with minimal hardware. For systems that do not need high-performance timers and counters, machine-mode software can trap accesses and implement all features in software. For systems that need high-performance timers and counters but are not concerned with obfuscating the underlying hardware counters, the delta-registers can be hardwired to zero and the raw counters made directly accessible to lower privilege modes. Systems that need both high performance accesses and wish to virtualize or obfuscate the timers and counters can provide full hardware support for the counter deltas.

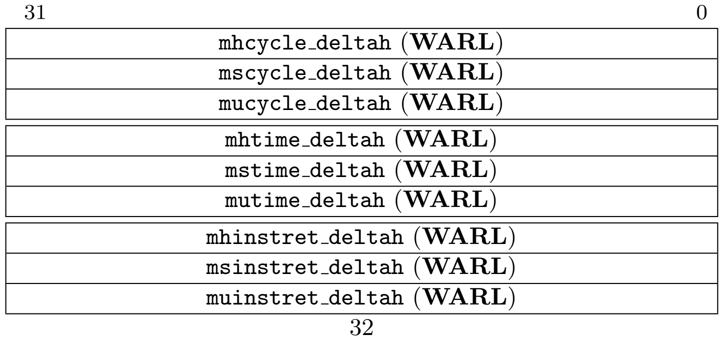

Figure 3.22: Upper 32 bits of machine counter-delta registers, RV32 only.

3.1.18 Machine Scratch Register (`mscratch`)

The `mscratch` register is an XLEN-bit read/write register dedicated for use by machine mode. Typically, it is used to hold a pointer to a machine-mode hart-local context space and swapped with a user register upon entry to an M-mode trap handler.

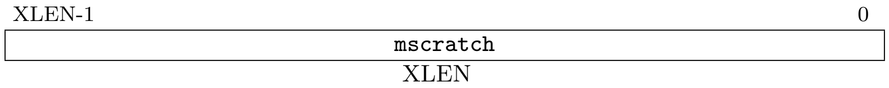

Figure 3.23: Machine-mode scratch register.

The MIPS ISA allocated two user registers (`k0`/`k1`) for use by the operating system. Although the MIPS scheme provides a fast and simple implementation, it also reduces available user registers, and does not scale to further privilege levels, or nested traps. It can also require both registers are cleared before returning to user level to avoid a potential security hole and to provide deterministic debugging behavior.

The RISC-V user ISA was designed to support many possible privileged system environments and so we did not want to infect the user-level ISA with any OS-dependent features. The RISC-V CSR swap instructions can quickly save/restore values to the `mscratch` register. Unlike the MIPS design, the OS can rely on holding a value in the `mscratch` register while the user context is running.

3.1.19 Machine Exception Program Counter (`mepc`)

`mepc` is an XLEN-bit read/write register formatted as shown in Figure 3.24. The low bit of `mepc` (`mepc[0]`) is always zero. On implementations that do not support instruction-set extensions with 16-bit instruction alignment, the two low bits (`mepc[1:0]`) are always zero.

The `mepc` register can never hold a PC value that would cause an instruction-address-misaligned exception.

When a trap is taken, `mepc` is written with the virtual address of the instruction that encountered the exception.

XLEN-1	0
mepc	
XLEN	

Figure 3.24: Machine exception program counter register.

3.1.20 Machine Cause Register (`mcause`)

The `mcause` register is an XLEN-bit read-write register formatted as shown in Figure 3.25. The Interrupt bit is set if the trap was caused by an interrupt. The Exception Code field contains a code identifying the last exception. Table 3.6 lists the possible machine-level exception codes. The Exception Code is an **WLRL** field, so is only guaranteed to hold supported exception codes.

XLEN-1	XLEN-2	0
Interrupt	Exception Code (**WLRL**)	
1	XLEN-1	

Figure 3.25: Machine Cause register `mcause`.

We do not distinguish privileged instruction exceptions from illegal opcode exceptions. This simplifies the architecture and also hides details of which higher-privilege instructions are supported by an implementation. The privilege level servicing the trap can implement a policy on whether these need to be distinguished, and if so, whether a given opcode should be treated as illegal or privileged.

Interrupts can be separated from other traps with a single branch on the sign of the `mcause` *register value. A single shift left can remove the interrupt bit and scale the exception codes to index into a trap vector table.*

3.1.21 Machine Bad Address (`mbadaddr`) Register

`mbadaddr` is an XLEN-bit read-write register formatted as shown in Figure 3.26. When an instruction-fetch, load, store, or AMO address-misaligned or access exception occurs, `mbadaddr` is written with the faulting address. The value in `mbadaddr` is undefined for other exceptions.

XLEN-1	0
mbadaddr	
XLEN	

Figure 3.26: Machine bad address register.

For instruction-fetch access faults on RISC-V systems with variable-length instructions, `mbadaddr` will point to the portion of the instruction that caused the fault while `mepc` will point to the beginning of the instruction.

Interrupt	Exception Code	Description
1	0	User software interrupt
1	1	Supervisor software interrupt
1	2	Hypervisor software interrupt
1	3	Machine software interrupt
1	4	User timer interrupt
1	5	Supervisor timer interrupt
1	6	Hypervisor timer interrupt
1	7	Machine timer interrupt
1	8	User external interrupt
1	9	Supervisor external interrupt
1	10	Hypervisor external interrupt
1	11	Machine external interrupt
1	≥12	*Reserved*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	Environment call from H-mode
0	11	Environment call from M-mode
0	≥12	*Reserved*

Table 3.6: Machine cause register (`mcause`) values after trap.

3.2 Machine-Mode Privileged Instructions

3.2.1 Trap-Return Instructions

Instructions to return from trap are encoded under the PRIV minor opcode.

31 20	19 15	14 12	11 7	6 0
funct12	rs1	funct3	rd	opcode
12	5	3	5	7
MRET/HRET/SRET/URET	0	PRIV	0	SYSTEM

To return after handling a trap, there are separate trap return instructions per privilege level: MRET, HRET, SRET, and URET. MRET is always provided, while HRET and SRET must be provided if the respective privilege mode is supported. URET is only provided if user-mode traps are supported. An xRET instruction can be executed in privilege mode x or higher, where executing a lower-privilege xRET instruction will pop the relevant lower-privilege interrupt enable and privilege mode stack. In addition to manipulating the privilege stack as described in Section 3.1.7, xRET sets the `pc` to the value stored in the x`epc` register.

Previously, there was only a single ERET instruction (which was also earlier known as SRET). To support the addition of user-level interrupts, we needed to add a separate URET instruction to continue to allow classic virtualization of OS code using the ERET instruction. It then became more orthogonal to support a different x*RET instruction per privilege level (which also enables virtualization of a hypervisor at supervisor level).*

3.2.2 Wait for Interrupt

The Wait for Interrupt instruction (WFI) provides a hint to the implementation that the current hart can be stalled until an interrupt might need servicing. Execution of the WFI instruction can also be used to inform the hardware platform that suitable interrupts should preferentially be routed to this hart. WFI is available in all of the supported S, H, and M privilege modes, and optionally available to U-mode for implementations that support U-mode interrupts.

31 20	19 15	14 12	11 7	6 0
funct12	rs1	funct3	rd	opcode
12	5	3	5	7
WFI	0	PRIV	0	SYSTEM

If an enabled interrupt is present or later becomes present while the hart is stalled, the interrupt exception will be taken on the following instruction, i.e., execution resumes in the trap handler and `mepc` = `pc` + 4.

The following instruction takes the interrupt exception and trap, so that a simple return from the trap handler will execute code after the WFI instruction.

The WFI instruction is just a hint, and a legal implementation is to implement WFI as a NOP.

If the implementation does not stall the hart on execution of the instruction, then the interrupt will be taken on some instruction in the idle loop containing the WFI, and on a simple return from the handler, the idle loop will resume execution.

We have removed the earlier requirement that implementations ignore the rs1 *and* rd *fields, so non-zero values in these fields should now raise illegal instruction exceptions.*

The WFI instruction can also be executed when interrupts are disabled. The operation of WFI must be unaffected by the global interrupt bits in `mstatus` (MIE/HIE/SIE/UIE) (i.e., the hart must resume if a locally enabled interrupt becomes pending), but should honor the individual interrupt enables (e.g, MTIE) (i.e., implementations should avoid resuming the hart if the interrupt is pending but not individually enabled). WFI is also required to resume execution for locally enabled interrupts pending at any privilege level, regardless of the global interrupt enable at each privilege level.

If the event that causes the hart to resume execution does not cause an interrupt to be taken, execution will resume at `pc` + 4, and software must determine what action to take, including looping back to repeat the WFI if there was no actionable event.

By allowing wakeup when interrupts are disabled, an alternate entry point to an interrupt handler can be called that does not require saving the current context, as the current context can be saved or discarded before the WFI is executed.

As implementations are free to implement WFI as a NOP, software must explicitly check for any relevant pending but disabled interrupts in the code following an WFI, and should loop back to the WFI if no suitable interrupt was detected. The `mip`, `hip`, `sip`, *or* `uip` *registers can be interrogated to determine the presence of any interrupt in machine, hypervisor, supervisor, or user mode respectively.*

The operation of WFI is unaffected by the delegation register settings.

WFI is defined so that an implementation can trap into a higher privilege mode, either immediately on encountering the WFI or after some interval to initiate a machine-mode transition to a lower power state, for example.

The same "wait-for-event" template might be used for possible future extensions that wait on memory locations changing, or message arrival.

3.3 Reset

Upon reset, a hart's privilege mode is set to M. The `mstatus` fields MIE and MPRV are reset to 0, and the VM field is reset to Mbare. The `pc` is set to an implementation-defined reset vector. The `mcause` register is set to a value indicating the cause of the reset. All other hart state is undefined.

The `mcause` values after reset have implementation-specific interpretation, but the value 0 should be returned on implementations that do not distinguish different reset conditions. Implementations that distinguish different reset conditions should only use 0 to indicate the most complete reset (e.g., hard reset).

Some designs may have multiple causes of reset (e.g., power-on reset, external hard reset, brownout detected, watchdog timer elapse, sleep-mode wakeup), which machine-mode software and debuggers may wish to distinguish.

`mcause` *reset values may alias* `mcause` *values following synchronous exceptions. There is no ambiguity in this overlap, since on reset the* `pc` *is set to a different value than on other traps.*

3.4 Non-Maskable Interrupts

Non-maskable interrupts (NMIs) are only used for hardware error conditions, and cause an immediate jump to an implementation-defined NMI vector running in M-mode regardless of the state of a hart's interrupt enable bits. The `mepc` register is written with the address of the next instruction to be executed at the time the NMI was taken, and `mcause` is set to a value indicating the source of the NMI. The NMI can thus overwrite state in an active machine-mode interrupt handler.

The values written to `mcause` on an NMI are implementation-defined, but a value of 0 is reserved to mean "unknown cause" and implementations that do not distinguish sources of NMIs via the `mcause` register should return 0.

Unlike resets, NMIs do not reset processor state, enabling diagnosis, reporting, and possible containment of the hardware error.

3.5 Physical Memory Attributes

The physical memory map for a complete system includes various address ranges, some corresponding to memory regions, some to memory-mapped control registers, and some to empty holes in the address space. Some memory regions might not support reads, writes, or execution; some might not support subword or subblock accesses; some might not support atomic operations; and some might not support cache coherence or might have different memory models. Similarly, memory-mapped control registers vary in their supported access widths, support for atomic operations, and whether read and write accesses have associated side effects. In RISC-V systems, these properties and capabilities of each region of the machine's physical address space are termed *physical memory attributes* (PMAs). This section describes RISC-V PMA terminology and how RISC-V systems implement and check PMAs.

PMAs are inherent properties of the underlying hardware and rarely change during system operation. Unlike physical memory protection values described in Section 3.6, PMAs do not vary by execution context. The PMAs of some memory regions are fixed at chip design time—for example, for an on-chip ROM. Others are fixed at board design time, depending, for example, on which other chips are connected to off-chip buses. Off-chip buses might also support devices that could be changed on every power cycle (cold pluggable) or dynamically while the system is running (hot pluggable). Some devices might be configurable at run time to support different uses that imply different PMAs—for example, an on-chip scratchpad RAM might be cached privately by one core in one end-application, or accessed as a shared non-cached memory in another end-application.

Most systems will require that at least some PMAs are dynamically checked in hardware later in the execution pipeline after the physical address is known, as some operations will not be supported at all physical memory addresses, and some operations require knowing the current setting of a configurable PMA attribute. While many other systems specify some PMAs in the virtual memory page tables and use the TLB to inform the pipeline of these properties, this approach injects platform-specific information into a virtualized layer and can cause system errors unless attributes are correctly initialized in each page-table entry for each physical memory region. In addition, the available page sizes might not be optimal for specifying attributes in the physical memory space, leading to address-space fragmentation and inefficient use of expensive TLB entries.

For RISC-V, we separate out specification and checking of PMAs into a separate hardware structure, the *PMA checker*. In many cases, the attributes are known at system design time for each physical address region, and can be hardwired into the PMA checker. Where the attributes are run-time configurable, platform-specific memory-mapped control registers can be provided to specify these attributes at a granularity appropriate to each region on the platform (e.g., for an on-chip SRAM that can be flexibly divided between cacheable and uncacheable uses). PMAs are checked for any access to physical memory, including accesses that have undergone virtual to physical memory translation. To aid in system debugging, we strongly recommend that, where possible, RISC-V processors precisely trap physical memory accesses that fail PMA checks. Precise PMA traps might not always be possible, for example, when probing a legacy bus architecture that uses access failures as part of the discovery mechanism. In this case, error responses from slave devices will be reported as imprecise bus-error interrupts.

PMAs must also be readable by software to correctly access certain devices or to correctly configure other hardware components that access memory, such as DMA engines. As PMAs are tightly tied to a given physical platform's organization, many details are inherently platform-specific, as is the means by which software can learn the PMA values for a platform. The configuration string (Chapter 8) can encode PMAs for on-chip devices and might also describe on-chip controllers for off-chip buses that can be dynamically interrogated to discover attached device PMAs. Some devices, particularly legacy buses, do not support discovery of PMAs and so will give error responses or time out if an unsupported access is attempted. Typically, platform-specific machine-mode code will extract PMAs and ultimately present this information to higher-level less-privileged software using some standard representation.

Where platforms support dynamic reconfiguration of PMAs, an interface will be provided to set the attributes by passing requests to a machine-mode driver that can correctly reconfigure the platform. For example, switching cacheability attributes on some memory regions might involve platform-specific operations, such as cache flushes, that are available only to machine-mode.

3.5.1 Main Memory versus I/O versus Empty Regions

The most important characterization of a given memory address range is whether it holds regular main memory, or I/O devices, or is empty. Regular main memory is required to have a number of properties, specified below, whereas I/O devices can have a much broader range of attributes. Memory regions that do not fit into regular main memory, for example, device scratchpad RAMs, are categorized as I/O regions. Empty regions are also classified as I/O regions but with attributes specifying that no accesses are supported.

3.5.2 Supported Access Type PMAs

Access types specify which access widths, from 8-bit byte to long multi-word burst, are supported, and also whether misaligned accesses are supported for each access width.

Although software running on a RISC-V hart cannot directly generate bursts to memory, software might have to program DMA engines to access I/O devices and might therefore need to know which access sizes are supported.

Main memory regions always support read, write, and execute of all access widths required by the attached devices.

In some cases, the design of a processor or device accessing main memory might support other widths, but must be able to function with the types supported by the main memory.

I/O regions can specify which combinations of read, write, or execute accesses to which data widths are supported.

3.5.3 Atomicity PMAs

Atomicity PMAs describes which atomic instructions are supported in this address region. Main memory regions must support the atomic operations required by the processors attached. I/O regions may only support a subset or none of the processor-supported atomic operations.

Support for atomic instructions is divided into two categories: *LR/SC* and *AMOs*. Within AMOs, there are four levels of support: *AMONone*, *AMOSwap*, *AMOLogical*, and *AMOArithmetic*. AMONone indicates that no AMO operations are supported. AMOSwap indicates that only `amoswap` instructions are supported in this address range. AMOLogical indicates that swap instructions plus all the logical AMOs (`amoand`, `amoor`, `amoxor`) are supported. AMOArithmetic indicates that all RISC-V AMOs are supported. For each level of support, naturally aligned AMOs of a given width are supported if the underlying memory region supports reads and writes of that width.

AMO Class	Supported Operations
AMONone	*None*
AMOSwap	`amoswap`
AMOLogical	above + `amoand`, `amoor`, `amoxor`
AMOArithmetic	above + `amoadd`, `amomin`, `amomax`, `amominu`, `amomaxu`

Table 3.7: Classes of AMOs supported by I/O regions. Main memory regions must always support all AMOs required by the processor.

We recommend providing at least AMOLogical support for I/O regions where possible. Most I/O regions will not support LR/SC accesses, as these are most conveniently built on top of a cache-coherence scheme.

3.5.4 Memory-Ordering PMAs

Regions of the address space are classified as either *main memory* or *I/O* for the purposes of ordering by the FENCE instruction and atomic-instruction ordering bits.

Accesses by one hart to main memory regions are observable not only by other harts but also by other devices with the capability to initiate requests in the main memory system (e.g., DMA engines). Main memory regions always have the standard RISC-V relaxed memory model.

Accesses by one hart to the I/O space are observable not only by other harts and bus mastering devices, but also by targeted slave I/O devices. Within I/O, regions may further be classified as implementing either *relaxed* or *strong* ordering. A relaxed I/O region has no ordering guarantees on how memory accesses made by one hart are observable by different harts or I/O devices beyond those enforced by FENCE and AMO instructions. A strongly ordered I/O region ensures that all accesses made by a hart to that region are only observable in program order by all other harts or I/O devices.

Each strongly ordered I/O region specifies a numbered ordering channel, which is a mechanism by which ordering guarantees can be provided between different I/O regions. Channel 0 is used to indicate point-to-point strong ordering only, where only accesses by the hart to the single associated I/O region are strongly ordered.

Channel 1 is used to provide global strong ordering across all I/O regions. Any accesses by a hart to any I/O region associated with channel 1 can only be observed to have occurred in program order by all other harts and I/O devices, including relative to accesses made by that hart to relaxed I/O regions or strongly ordered I/O regions with different channel numbers. In other words, any access to a region in channel 1 is equivalent to executing a `fence io,io` instruction before and after the instruction.

Other larger channel numbers provide program ordering to accesses by that hart across any regions with the same channel number.

Systems might support dynamic configuration of ordering properties on each memory region.

Strong ordering can be used to improve compatibility with legacy device driver code, or to enable increased performance compared to insertion of explicit ordering instructions when the implementation is known to not reorder accesses.

Local strong ordering (channel 0) is the default form of strong ordering as it is often straightforward to provide if there is only a single in-order communication path between the hart and the I/O device.

Generally, different strongly ordered I/O regions can share the same ordering channel without additional ordering hardware if they share the same interconnect path and the path does not reorder requests.

3.5.5 Coherence and Cacheability PMAs

Coherence is a property defined for a single physical address, and indicates that writes to that address by one agent will eventually be made visible to other agents in the system. Coherence is not to be confused with the memory consistency model of a system, which defines what values a memory read can return given the previous history of reads and writes to the entire memory system. In RISC-V platforms, the use of hardware-incoherent regions is discouraged due to software complexity, performance, and energy impacts.

The cacheability of a memory region should not affect the software view of the region except for differences reflected in other PMAs, such as main memory versus I/O classification, memory ordering, supported accesses and atomic operations, and coherence. For this reason, we treat cacheability as a platform-level setting managed by machine-mode software only.

Where a platform supports configurable cacheability settings for a memory region, a platform-specific machine-mode routine will change the settings and flush caches if necessary, so the system is only incoherent during the transition between cacheability settings. This transitory state should not be visible to lower privilege levels.

We categorize RISC-V caches into three types: master-private, shared, *and* slave-private. *Master-private caches are attached to a single master agent, i.e., one that issues read/write requests to the memory system. Shared caches are located inbetween masters and slaves and may be hierarchically organized. Slave-private caches do not impact coherence, as they are local to a single slave and do not affect other PMAs at a master, so are not considered further here. We use* private cache *to mean a master-private cache in the following section, unless explicitly stated otherwise.*

Coherence is straightforward to provide for a shared memory region that is not cached by any agent. The PMA for such a region would simply indicate it should not be cached in a private or shared cache.

Coherence is also straightforward for read-only regions, which can be safely cached by multiple agents without requiring a cache-coherence scheme. The PMA for this region would indicate that it can be cached, but that writes are not supported.

Some read-write regions might only be accessed by a single agent, in which case they can be cached privately by that agent without requiring a coherence scheme. The PMA for such regions would indicate they can be cached. The data can also be cached in a shared cache, as other agents should not access the region.

If an agent can cache a read-write region that is accessible by other agents, whether caching or non-caching, a cache-coherence scheme is required to avoid use of stale values. In regions lacking hardware cache coherence (hardware-incoherent regions), cache coherence can be implemented entirely in software, but software coherence schemes are notoriously difficult to implement correctly and often have severe performance impacts due to the need for conservative software-directed cache-flushing. Hardware cache-coherence schemes require more complex hardware and can impact performance due to the cache-coherence probes, but are otherwise invisible to software.

For each hardware cache-coherent region, the PMA would indicate that the region is coherent and which hardware coherence controller to use if the system has multiple coherence controllers. For some systems, the coherence controller might be an outer-level shared cache, which might itself access further outer-level cache-coherence controllers hierarchically.

Most memory regions within a platform will be coherent to software, because they will be fixed as either uncached, read-only, hardware cache-coherent, or only accessed by one agent.

3.5.6 Idempotency PMAs

Idempotency PMAs describe whether reads and writes to an address region are idempotent. Main memory regions are assumed to be idempotent. For I/O regions, idempotency on reads and writes can be specified separately (e.g., reads are idempotent but writes are not). If accesses are non-idempotent, i.e., there is potentially a side effect on any read or write access, then speculative or redundant accesses must be avoided.

For the purposes of defining the idempotency PMAs, changes in observed memory ordering created by redundant accesses are not considered a side effect.

While hardware should always be designed to avoid speculative or redundant accesses to memory regions marked as non-idempotent, it is also necessary to ensure software or compiler optimizations do not generate spurious accesses to non-idempotent memory regions.

3.6 Physical Memory Protection

To support secure processing and contain faults, it is desirable to limit the physical addresses accessible by a lower-privilege context running on a hart. A physical memory protection (PMP) unit can be provided, with per-hart machine-mode control registers to allow physical memory access privileges (read, write, execute) to be specified for each physical memory region. The PMP values are checked in parallel with the PMA checks described in Section 3.5.

The granularity and encoding of the PMP access control settings are platform-specific, and there might be different granularities and encodings of permissions for different physical memory regions on a single platform. Certain regions' privileges can be hardwired—for example, some regions might only ever be visible in machine mode but no lower-privilege layers.

Platforms vary widely in demands for physical memory protection, and so we defer detailed *design of PMP structures to each platform. Some PMP designs might just employ a few CSRs to protect a small number of physical memory segments, while others might employ memory-resident protection tables with a protection-table cache indexed by a protection-table base register to protect large physical memory spaces with fine granularity. Systems with a protection-table base register will usually also provide a physical protection domain ID (PDID) register to denote the current physical protection domain.*

PMP checks are applied to all accesses when the hart is running in H, S, or U modes, and for loads and stores when the MPRV bit is set in the `mstatus` register and the MPP field in the `mstatus` register contains H, S, or U. PMP violations will always be trapped precisely at the processor.

3.7 Mbare addressing environment

The Mbare environment is entered at reset, or can be entered at any time thereafter by writing 0 to the VM field in the `mstatus` register.

In the Mbare environment all virtual addresses are converted with no translation into physical addresses, with truncation of any excess high-order bits. Physical memory protection, as described in Section 3.6, can be used to constrain accesses by lower-privilege modes.

3.8 Base-and-Bound environments

This section describes the Mbb virtualization environment, which provides a base-and-bound translation and protection scheme. There are two variants of base-and-bound, Mbb and Mbbid, depending on whether there is a single base-and-bound (Mbb) or separate base-and-bounds for instruction fetches and data accesses (Mbbid). This simple translation and protection scheme has the advantage of low complexity and deterministic high performance, as there are never any TLB misses during operation.

3.8.1 Mbb: Single Base-and-Bound registers (`mbase`, `mbound`)

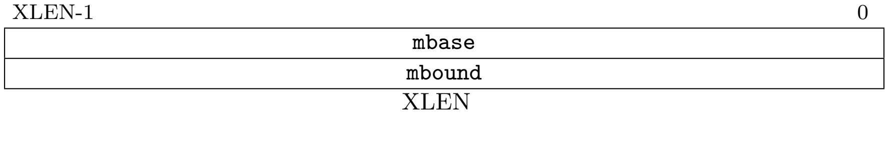

Figure 3.27: Single Base-and-Bound Registers.

The simpler Mbb system has a single base `mbase` and single bound `mbound` register. Mbb is enabled by writing the value 1 to the VM field in the `mstatus` register.

The base-and-bound registers define a contiguous virtual-address segment beginning at virtual address 0 with a length given in bytes by the value in `mbound`. This virtual address segment is mapped to a contiguous physical address segment starting at the physical address given in the `mbase` register.

When Mbb is in operation, all lower-privilege mode (U, S, H) instruction-fetch addresses and data addresses are translated by adding the value of `mbase` to the virtual address to obtain the physical address. Simultaneously, the virtual address is compared against the value in the bound register. An address fault exception is generated if the virtual address is equal to or greater than the virtual address limit held in the `mbound` register.

Machine-mode instruction fetch and data accesses are not translated or checked in Mbb (except for loads and stores when the MPRV bit is set in `mstatus`), so machine-mode effective addresses are treated as physical addresses.

3.8.2 Mbbid: Separate Instruction and Data Base-and-Bound registers

XLEN-1   0
`mibase`
`mibound`
`mdbase`
`mdbound`
XLEN

Figure 3.28: Separate instruction and data base-and-bound registers.

The Mbbid scheme separates the virtual address segments for instruction fetches and data accesses to allow a single physical instruction segment to be shared by two or more user-level virtual address spaces while a separate data segment is allocated to each. Mbbid is enabled by writing 2 to the VM field of `mstatus` register.

The split instruction and data base-and-bounds scheme was famously used on Cray supercomputers, where it avoids most runtime overheads related to translation and protection provided the segments fit in physical memory.

The `mibase` and `mibound` registers define the physical start address and length of the instruction segment respectively, while `mdbase` and `mdbound` specify the physical start address and length of the data segment respectively.

The data virtual address segment begins at address 0, while the instruction virtual address segment begins half way through the virtual address space, at an address given by a leading 1 following by XLEN-1 trailing zeros (e.g., `0x8000_0000` for 32-bit address space systems). The virtual addresses of lower privilege-mode instruction fetches are first checked to ensure their high bit is set; if not, an exception is generated. The high bit is subsequently treated as zero when adding the base to the virtual address and when checking the bound.

The data and instruction virtual address segments should not overlap, and we felt it more important to preserve the potential of zero page data accesses (using a 12-bit offset from register `x0`*) than to support instruction entry points using JALR with* `x0`*. In particular, a single JAL can directly access all of a 2 MiB code segment.*

To simplify linking, the instruction virtual address segment start address should be constant independent of the length of the complete binary. Placing at the midpoint of virtual memory minimizes the circuitry needed to separate the two segments.

Systems that provide Mbbid must also provide Mbb. Writes to the CSR addresses corresponding to `mbase` should write the same value to `mibase` & `mdbase`, and writes to `mbound` should write the same value to `mibound` & `mdbound` to provide compatible behavior. Reads of `mbase` should return the value in `mdbase` and reads of `mbound` should return the value in `mdbound`. When VM is set to Mbb, instruction fetches no longer check the high bit of the virtual address, and no longer reset the high bit to zero before adding base and checking bound.

While the split scheme allows a single physical instruction segment to be shared across multiple user process instances, it also effectively prevents the instruction segment from being written by the user program (data stores are translated separately) and prevents execution of instructions from the data segment (instruction fetches are translated separately). These restrictions can prevent some forms of security attack.

On the other hand, many modern programming systems require, or benefit from, some form of runtime-generated code, and so these should use the simpler Mbb mode with a single segment, which is partly why supporting this mode is required if providing Mbbid.

Chapter 4

Supervisor-Level ISA

This chapter describes the RISC-V supervisor-level architecture, which contains a common core that is used with various supervisor-level address translation and protection schemes. Supervisor-mode always operates inside a virtual memory scheme defined by the VM field in the machine-mode `mstatus` register. Supervisor-level code is written to a given VM scheme, and cannot change the VM scheme in use.

Supervisor-level code relies on a supervisor execution environment to initialize the environment and enter the supervisor code at an entry point defined by the system binary interface (SBI). The SBI also defines function entry points that provide supervisor environment services for supervisor-level code.

Supervisor mode is deliberately restricted in terms of interactions with underlying physical hardware, such as physical memory and device interrupts, to support clean virtualization. A more conventional virtualization-unfriendly operating system can be ported while retaining a protected M-mode environment by using M-mode to unprotect selected physical memory regions for access by the supervisor, and by delegating selected device interrupts to S-mode.

4.1 Supervisor CSRs

A number of CSRs are provided for the supervisor.

The supervisor should only view CSR state that should be visible to a supervisor-level operating system. In particular, there is no information about the existence (or non-existence) of higher privilege levels (hypervisor or machine) visible in the CSRs accessible by the supervisor.

Many supervisor CSRs are a subset of the equivalent machine-mode CSR, and the machine-mode chapter should be read first to help understand the supervisor-level CSR descriptions.

4.1.1 Supervisor Status Register (`sstatus`)

The `sstatus` register is an XLEN-bit read/write register formatted as shown in Figure 4.1. The `sstatus` register keeps track of the processor's current operating state.

XLEN-1	XLEN-2 19	18	17	16 15	14 13	12 9	8	7 6	5	4	3 2	1	0
SD	0	PUM	0	XS[1:0]	FS[1:0]	0	SPP	0	SPIE	UPIE	0	SIE	UIE
1	XLEN-20	1	1	2	2	4	1	2	1	1	2	1	1

Figure 4.1: Supervisor-mode status Register.

The SPP bit indicates the privilege level at which a hart was executing before entering supervisor mode. When a trap is taken, SPP is set to 0 if the trap originated from user mode, or 1 otherwise. When an SRET instruction (see Section 3.2.1) is executed to return from the trap handler, the privilege level is set to user mode if the SPP bit is 0, or supervisor mode if the SPP bit is 1; SPP is then set to 0.

The SIE bit enables or disables all interrupts in supervisor mode. When SIE is clear, interrupts are not taken while in supervisor mode. When the hart is running in user-mode, the value in SIE is ignored, and supervisor-level interrupts are enabled. The supervisor can disable indivdual interrupt sources using the `sie` register.

The SPIE bit indicates whether supervisor-level interrupts were enabled before entering supervisor mode. When a trap is taken, SPIE is set to SIE and SIE is set to 0. When an SRET instruction is executed, SIE is set to SPIE, and SPIE is set to 1.

The UIE bit enables or disables user-mode interrupts. User-level interrupts are enabled only if UIE is set and the hart is running in user-mode. The UPIE bit indicates whether user-level interrupts were enabled prior to taking a user-level trap. When a URET instruction is executed, UIE is set to UPIE, and UPIE is set to 1. User-level interrupts are optional. If omitted, the UIE and UPIE bits are hardwired to zero.

The `sstatus` *register is a subset of the* `mstatus` *register. In a straightforward implementation, reading or writing any field in* `sstatus` *is equivalent to reading or writing the homonymous field in* `mstatus`*.*

4.1.2 Memory Privilege in `sstatus` Register

The PUM (Protect User Memory) bit modifies the privilege with which S-mode loads, stores, and instruction fetches access virtual memory. When PUM=0, translation and protection behave as normal. When PUM=1, S-mode memory accesses to pages that are accessible by U-mode (U=1 in Figure 4.19) will fault. PUM has no effect when executing in U-mode.

The PUM mechanism prevents supervisor software from inadvertently accessing user memory. Operating systems can execute the majority of code with PUM set; the few code segments that should access user memory can temporarily clear PUM.

4.1.3 Supervisor Trap Vector Base Address Register (`stvec`)

The `stvec` register is an XLEN-bit read/write register that holds the base address of the S-mode trap vector. When an exception occurs, the `pc` is set to `stvec`. The `stvec` register is always aligned to a 4-byte boundary.

XLEN-1 2	1 0
`Trap Vector Base Address`	0
XLEN-2	2

Figure 4.2: Supervisor trap vector base address register (`stvec`).

4.1.4 Supervisor Interrupt Registers (`sip` and `sie`)

The `sip` register is an XLEN-bit read/write register containing information on pending interrupts, while `sie` is the corresponding XLEN-bit read/write register containing interrupt enable bits.

XLEN-1 10	9	8	7 6	5	4	3 2	1	0
0	SEIP	UEIP	0	STIP	UTIP	0	SSIP	USIP
XLEN-10	1	1	2	1	1	2	1	1

Figure 4.3: Supervisor interrupt-pending register (`sip`).

XLEN-1 10	9	8	7 6	5	4	3 2	1	0
0	SEIE	UEIE	0	STIE	UTIE	0	SSIE	USIE
XLEN-10	1	1	2	1	1	2	1	1

Figure 4.4: Supervisor interrupt-enable register (`sie`).

Three types of interrupts are defined: software interrupts, timer interrupts, and external interrupts. A supervisor-level software interrupt is triggered on the current hart by writing 1 to its supervisor software interrupt-pending (SSIP) bit in the `sip` register. A pending supervisor-level software interrupt can be cleared by writing 0 to the SSIP bit in `sip`. Supervisor-level software interrupts are disabled when the SSIE bit in the `sie` register is clear.

Interprocessor interrupts are sent to other harts by means of SBI calls, which will ultimately cause the SSIP bit to be set in the recipient hart's `sip` register.

A user-level software interrupt is triggered on the current hart by writing 1 to its user software interrupt-pending (USIP) bit in the `sip` register. A pending user-level software interrupt can be cleared by writing 0 to the USIP bit in `sip`. User-level software interrupts are disabled when the USIE bit in the `sie` register is clear. If user-level interrupts are not supported, USIP and USIE are hardwired to zero.

All bits besides SSIP and USIP in the `sip` register are read-only.

A supervisor-level timer interrupt is pending if the STIP bit in the `sip` register is set. Supervisor-level timer interrupts are disabled when the STIE bit in the `sie` register is clear. An SBI call to the SEE may be used to clear the pending timer interrupt.

A user-level timer interrupt is pending if the UTIP bit in the `sip` register is set. User-level timer interrupts are disabled when the UTIE bit in the `sie` register is clear. If user-level interrupts are supported, the ABI should provide a facility for scheduling timer interrupts in terms of real-time counter values. If user-level interrupts are not supported, UTIP and UTIE are hardwired to zero.

A supervisor-level external interrupt is pending if the SEIP bit in the `sip` register is set. Supervisor-level external interrupts are disabled when the SEIE bit in the `sie` register is clear. The SBI should provide facilities to mask, unmask, and query the cause of external interrupts.

A user-level external interrupt is pending if the UEIP bit in the `sip` register is set. User-level external interrupts are disabled when the UEIE bit in the `sie` register is clear. If user-level interrupts are not supported, UEIP and UEIE are hardwired to zero.

> *The* `sip` *and* `sie` *registers are subsets of the* `mip` *and* `mie` *registers. Reading any field, or writing any writable field, of* `sip`/`sie` *effects a read or write of the homonymous field of* `mip`/`mie`*.*

4.1.5 Supervisor Time Register (`stime`)

Supervisor mode includes a real-time counter, `stime`, which has 64-bit precision on all RV32, RV64, and RV128 systems. The SBI must provide a facility for determining the timebase of `stime`, which must run at a constant frequency.

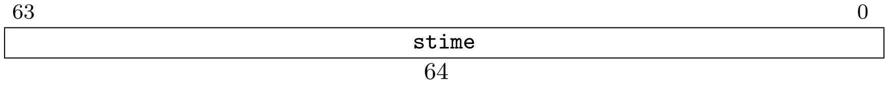

Figure 4.5: Supervisor time register.

On RV32 only, a read of the `stime` CSR returns the low 32 bits of `stime`, while the `stimeh` CSR accesses bits 63–32 of `stime`.

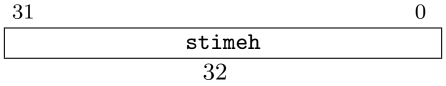

Figure 4.6: Upper 32 bits of supervisor time register, RV32 only.

The SBI must provide a facility for scheduling timer interrupts in terms of real-time counter values.

4.1.6 Supervisor Performance Counters (`scycle`, `sinstret`)

S-mode includes a basic performance measurement facility. The `scycle` CSR holds a count of the number of cycles the hart has executed since some arbitrary time in the past. The `sinstret` CSR holds a count of the number of instructions the hart has retired since some arbitrary time in the past. The `scycle` and `sinstret` registers have 64-bit precision on all RV32, RV64, and RV128 systems.

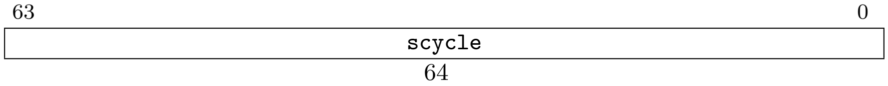

Figure 4.7: Supervisor cycle counter.

On RV32 only, the `scycle` and `sinstret` CSRs return the low 32 bits of the registers, while the `scycleh` and `sinstreth` CSRs return bits 63–32 of `scycle` and `sinstret`, respectively.

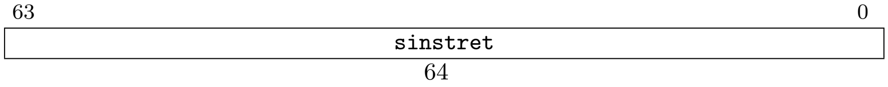

Figure 4.8: Supervisor instructions retired counter.

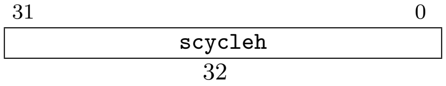

Figure 4.9: Upper 32 bits of supervisor cycle counter, RV32 only.

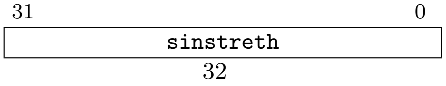

Figure 4.10: Upper 32 bits of supervisor instructions retired counter, RV32 only.

4.1.7 Supervisor Scratch Register (`sscratch`)

The `sscratch` register is an XLEN-bit read/write register, dedicated for use by the supervisor. Typically, `sscratch` is used to hold a pointer to the hart-local supervisor context while the hart is executing user code. At the beginning of a trap handler, `sscratch` is swapped with a user register to provide an initial working register.

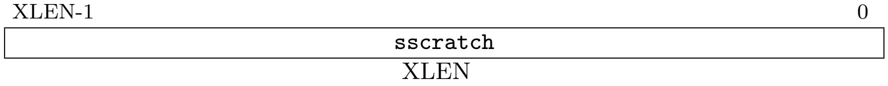

Figure 4.11: Supervisor Scratch Register.

4.1.8 Supervisor Exception Program Counter (`sepc`)

`sepc` is an XLEN-bit read/write register formatted as shown in Figure 4.12. The low bit of `sepc` (`sepc[0]`) is always zero. On implementations that do not support instruction-set extensions with 16-bit instruction alignment, the two low bits (`sepc[1:0]`) are always zero.

When a trap is taken, `sepc` is written with the virtual address of the instruction that encountered the exception.

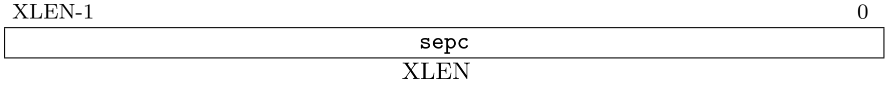

Figure 4.12: Supervisor exception program counter register.

4.1.9 Supervisor Cause Register (scause)

The scause register is an XLEN-bit read-write register formatted as shown in Figure 4.13. The Interrupt bit is set if the exception was caused by an interrupt. The Exception Code field contains a code identifying the last exception. Table 4.1 lists the possible exception codes for the current supervisor ISAs, in descending order of priority. The Exception Code is an **WLRL** field, so is only guaranteed to hold supported exception codes.

XLEN-1	XLEN-2 0
Interrupt	Exception Code (**WLRL**)
1	XLEN-1

Figure 4.13: Supervisor Cause register scause.

Interrupt	Exception Code	Description
1	0	User software interrupt
1	1	Supervisor software interrupt
1	2–3	*Reserved*
1	4	User timer interrupt
1	5	Supervisor timer interrupt
1	≥6	*Reserved*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	*Reserved*
0	5	Load access fault
0	6	AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call
0	≥9	*Reserved*

Table 4.1: Supervisor cause register (scause) values after trap.

4.1.10 Supervisor Bad Address (sbadaddr) Register

sbadaddr is an XLEN-bit read/write register formatted as shown in Figure 4.14. When an instruction fetch address-misaligned exception, or instruction fetch access exception, or AMO address-misaligned exception, or load or store access exception occurs, sbadaddr is written with the faulting address.

XLEN-1 0
sbadaddr
XLEN

Figure 4.14: Supervisor bad address register.

For instruction fetch access faults on RISC-V systems with variable-length instructions, `sbadaddr` will point to the portion of the instruction that caused the fault while `sepc` will point to the beginning of the instruction.

4.1.11 Supervisor Page-Table Base Register (`sptbr`)

The `sptbr` register is an XLEN-bit read/write register formatted as shown in Figure 4.15 for RV32 and Figure 4.16. The `sptbr` register is only present on systems supporting paged virtual-memory systems. This register holds the physical page number (PPN) of the root page table, i.e., its supervisor physical address divided by 4 KiB, and an address space identifier (ASID), which facilitates address-translation fences on a per-address-space basis.

The number of supervisor physical address bits is implementation-defined; any unimplemented address bits are hardwired to zero in the `sptbr` register. The number of ASID bits is also implementation-defined and may be zero. The number of implemented ASID bits may be determined by writing one to every bit position in the ASID field, then reading back the value in `sptbr` to see which bit positions in the ASID field hold a one.

We store the ASID and the page table base address in the same CSR to allow the pair to be changed atomically on a context switch. Swapping them non-atomically could pollute the old virtual address space with new translations, or vice-versa. This approach also slightly reduces the cost of a context switch.

31 22	21 0
ASID (**WARL**)	PPN (**WARL**)
10	22

Figure 4.15: RV32 Supervisor Page-Table Base Register `sptbr`.

Storing a PPN in `sptbr`*, rather than a physical address, supports physical address spaces larger than* 2^{XLEN}.

63 38	37 0
ASID (**WARL**)	PPN (**WARL**)
26	38

Figure 4.16: RV64 Supervisor Page-Table Base Register `sptbr`.

For many applications, the choice of page size has a substantial performance impact. A large page size increases TLB reach and loosens the associativity constraints on virtually-indexed, physically-tagged caches. At the same time, large pages exacerbate internal fragmentation, wasting physical memory and possibly cache capacity.

After much deliberation, we have settled on a conventional page size of 4 KiB for both RV32 and RV64. We expect this decision to ease the porting of low-level runtime software and device drivers. The TLB reach problem is ameliorated by transparent superpage support in modern operating systems [2]. Additionally, multi-level TLB hierarchies are quite inexpensive relative to the multi-level cache hierarchies whose address space they map.

4.2 Supervisor Instructions

In addition to the ECALL, EBREAK, and SRET instructions defined in Section 3.2.1, one new supervisor-level instruction is provided.

4.2.1 Supervisor Memory-Management Fence Instruction

31 20	19 15	14 12	11 7	6 0
funct12	rs1	funct3	rd	opcode
12	5	3	5	7
SFENCE.VM	vaddr	PRIV	0	SYSTEM

The supervisor memory-management fence instruction SFENCE.VM is used to synchronize updates to in-memory memory-management data structures with current execution. Instruction execution causes implicit reads and writes to these data structures; however, these implicit references are ordinarily not ordered with respect to loads and stores in the instruction stream. Executing an SFENCE.VM instruction guarantees that any stores in the instruction stream prior to the SFENCE.VM are ordered before all implicit references subsequent to the SFENCE.VM. Furthermore, executing an SFENCE.VM guarantees that any implicit writes caused by instructions prior to the SFENCE.VM are orderd before all loads and stores subsequent to the SFENCE.VM.

The SFENCE.VM is used to flush any local hardware caches related to address translation. It is specified as a fence rather than a TLB flush to provide cleaner semantics with respect to which instructions are affected by the flush operation and to support a wider variety of dynamic caching structures and memory-management schemes. SFENCE.VM is also used by higher privilege levels to synchronize page table writes and the address translation hardware.

Note the instruction has no effect on the translations of other RISC-V threads, which must be notified separately. One approach is to use 1) a local data fence to ensure local writes are visible globally, then 2) an interprocessor interrupt to the other thread, then 3) a local SFENCE.VM in the interrupt handler of the remote thread, and finally 4) signal back to originating thread that operation is complete. This is, of course, the RISC-V analog to a TLB shootdown. Alternatively, implementations might provide direct hardware support for remote TLB invalidation. TLB shootdowns are handled by an SBI call to hide implementation details.

The behavior of SFENCE.VM depends on the current value of the ASID field in the `sptbr` register. If ASID is nonzero, SFENCE.VM takes effect only for address translations in the current address space. If ASID is zero, SFENCE.VM affects address translations for all address spaces. In this case, it also affects *global* mappings, which are described in Section 4.5.1.

The register operand *rs1* contains an optional virtual address argument. If *rs1*=`x0`, the fence affects all virtual address translations and stores made to any level of the page tables.

For the common case that the translation data structures have only been modified for a single address mapping (i.e., one page) within a single leaf page table, *rs1* can specify a virtual address within that mapping to effect a translation fence for that mapping only. When $rs1 \neq x0$, the SFENCE.VM orders only stores to the leaf page table for the virtual address in *rs1*, and not any stores to other levels of the page table.

Simpler implementations can ignore the ASID value in `sptbr` *and the virtual address in* rs1 *and always perform a global fence.*

4.3 Supervisor Operation in Mbare Environment

When the Mbare environment is selected in the VM field of `mstatus` (Section 3.1.8), supervisor-mode virtual addresses are truncated and mapped directly to supervisor-mode physical addresses. Supervisor physical addresses are then checked using any physical memory protection structures (Section 3.6), before being directly converted to machine-level physical addresses.

4.4 Supervisor Operation in Base and Bounds Environments

When Mbb or Mbbid are selected in the VM field of `mstatus` (Section 3.1.8), supervisor-mode virtual addresses are translated and checked according to the appropriate machine-level base and bound registers. The resulting supervisor-level physical addresses are then checked using any physical memory protection structures (Section 3.6), before being directly converted to machine-level physical addresses.

4.5 Sv32: Page-Based 32-bit Virtual-Memory Systems

When Sv32 is written to the VM field in the `mstatus` register, the supervisor operates in a 32-bit paged virtual-memory system. Sv32 is supported on RV32 systems and is designed to include mechanisms sufficient for supporting modern Unix-based operating systems.

The initial RISC-V paged virtual-memory architectures have been designed as straightforward implementations to support existing operating systems. We have architected page table layouts to support a hardware page-table walker. Software TLB refills are a performance bottleneck on high-performance systems, and are especially troublesome with decoupled specialized coprocessors. An implementation can choose to implement software TLB refills using a machine-mode trap handler as an extension to M-mode.

4.5.1 Addressing and Memory Protection

Sv32 implementations support a 32-bit virtual address space, divided into 4 KiB pages. An Sv32 virtual address is partitioned into a virtual page number (VPN) and page offset, as shown in Figure 4.17. When Sv32 virtual memory mode is selected in the VM field of the `mstatus` register, supervisor virtual addresses are translated into supervisor physical addresses via a two-level page table. The 20-bit VPN is translated into a 22-bit physical page number (PPN), while the 12-bit page offset is untranslated. The resulting supervisor-level physical addresses are then checked using any physical memory protection structures (Sections 3.6), before being directly converted to machine-level physical addresses.

31 22	21 12	11 0
VPN[1]	VPN[0]	page offset
10	10	12

Figure 4.17: Sv32 virtual address.

33 22	21 12	11 0
PPN[1]	PPN[0]	page offset
12	10	12

Figure 4.18: Sv32 physical address.

31 20	19 10	9 8	7	6	5	4	3	2	1	0
PPN[1]	PPN[0]	*Reserved for software*	D	A	G	U	X	W	R	V
12	10	2	1	1	1	1	1	1	1	1

Figure 4.19: Sv32 page table entry.

Sv32 page tables consist of 2^{10} page-table entries (PTEs), each of four bytes. A page table is exactly the size of a page and must always be aligned to a page boundary. The physical page number of the root page table is stored in the `sptbr` register.

The PTE format for Sv32 is shown in Figures 4.19. The V bit indicates whether the PTE is valid; if it is 0, bits 31–1 of the PTE are don't-cares and may be used freely by software. The permission bits, R, W, and X, indicate whether the page is readable, writable, and executable, respectively. When all three are zero, the PTE is a pointer to the next level of the page table; otherwise, it is a leaf PTE. Writable pages must also be marked readable; the contrary combinations are reserved for future use. Table 4.2 summarizes the encoding of the permission bits.

X	W	R	Meaning
0	0	0	Pointer to next level of page table.
0	0	1	Read-only page.
0	1	0	*Reserved for future use.*
0	1	1	Read-write page.
1	0	0	Execute-only page.
1	0	1	Read-execute page.
1	1	0	*Reserved for future use.*
1	1	1	Read-write-execute page.

Table 4.2: Encoding of PTE R/W/X fields.

The U bit indicates whether the page is accessible to user mode. U-mode software may only access the page when U=1. If the PUM bit in the `sstatus` register is clear, supervisor mode software may also access pages with U=1. However, supervisor code normally operates with the PUM bit set, in which case, supervisor code will fault on accesses to user-mode pages.

An alternative PTE format would support different permissions for supervisor and user. We omitted this feature because it would be largely redundant with the PUM mechanism (see Section 4.1.2) and would require more encoding space in the PTE.

The G bit designates a *global* mapping. Global mappings are those that exist in all address spaces. For non-leaf PTEs, the global setting implies that all mappings in the subsequent levels of the page table are global. Note that failing to mark a global mapping as global merely reduces performance, whereas marking a non-global mapping as global is an error.

Global mappings were devised to reduce the cost of context switches. They need not be flushed from an implementation's address translation caches when an SFENCE.VM instruction is executed with a nonzero ASID value in `sptbr`*.*

Each leaf PTE maintains an accessed (A) and dirty (D) bit. When a virtual page is read, written, or fetched from, the implementation sets the A bit in the corresponding PTE. When a virtual page is written, the implementation additionally sets the D bit in the corresponding PTE. The PTE updates are exact and are observed in program order by the local hart. The ordering on loads and stores provided by FENCE instructions and the acquire/release bits on atomic instructions also orders the PTE updates associated with those loads and stores as observed by remote harts.

We have changed the behavior of the PTE updates to be exact and in program order on a hart. This significantly simplifies the specification, and can be implemented with high performance.

The A and D bits are never cleared by the implementation. If the supervisor software does not rely on accessed and/or dirty bits, e.g. if it does not swap memory pages to secondary storage or if the pages are being used to map I/O space, it should always set them to 1 in the PTE. The implementation can then avoid issuing memory accesses to set the bits.

Any level of PTE may be a leaf PTE, so in addition to 4 KiB pages, Sv32 supports 4 MiB *megapages*. A megapage must be virtually and physically aligned to a 4 MiB boundary.

4.5.2 Virtual Address Translation Process

A virtual address *va* is translated into a physical address *pa* as follows:

1. Let a be `sptbr`.$ppn \times$ PAGESIZE, and let $i =$ LEVELS $- 1$. (For Sv32, PAGESIZE=2^{12} and LEVELS=2.)
2. Let *pte* be the value of the PTE at address $a + va.vpn[i] \times$PTESIZE. (For Sv32, PTESIZE=4.)
3. If $pte.v = 0$, or if $pte.r = 0$ and $pte.w = 1$, stop and raise an access exception.
4. Otherwise, the PTE is valid. If $pte.r = 1$ or $pte.x = 1$, go to step 5. Otherwise, this PTE is a pointer to the next level of the page table. Let $i = i - 1$. If $i < 0$, stop and raise an access exception. Otherwise, let $a = pte.ppn \times$ PAGESIZE and go to step 2.
5. A leaf PTE has been found. Determine if the requested memory access is allowed by the $pte.r$, $pte.w$, and $pte.x$ bits. If not, stop and raise an access exception. Otherwise, the translation is successful. Set $pte.a$ to 1, and, if the memory access is a store, set $pte.d$ to 1. The translated physical address is given as follows:
   - $pa.pgoff = va.pgoff$.
   - If $i > 0$, then this is a superpage translation and $pa.ppn[i - 1 : 0] = va.vpn[i - 1 : 0]$.
   - $pa.ppn[\text{LEVELS} - 1 : i] = pte.ppn[\text{LEVELS} - 1 : i]$.

4.6 Sv39: Page-Based 39-bit Virtual-Memory System

This section describes a simple paged virtual-memory system designed for RV64 systems, which supports 39-bit virtual address spaces. The design of Sv39 follows the overall scheme of Sv32, and this section details only the differences between the schemes.

4.6.1 Addressing and Memory Protection

Sv39 implementations support a 39-bit virtual address space, divided into 4 KiB pages. An Sv39 address is partitioned as shown in Figure 4.20. Load and store effective addresses, which are 64 bits, must have bits 63–39 all equal to bit 38, or else an access fault will occur. The 27-bit VPN is translated into a 38-bit PPN via a three-level page table, while the 12-bit page offset is untranslated.

38 30	29 21	20 12	11 0
VPN[2]	VPN[1]	VPN[0]	page offset
9	9	9	12

Figure 4.20: Sv39 virtual address.

49 30	29 21	20 12	11 0
PPN[2]	PPN[1]	PPN[0]	page offset
20	9	9	12

Figure 4.21: Sv39 physical address.

63 48	47 28	27 19	18 10	9 8	7	6	5	4	3	2	1	0
Reserved	PPN[2]	PPN[1]	PPN[0]	*Reserved for SW*	D	A	G	U	X	W	R	V
16	20	9	9	2	1	1	1	1	1	1	1	1

Figure 4.22: Sv39 page table entry.

Sv39 page tables contain 2^9 page table entries (PTEs), eight bytes each. A page table is exactly the size of a page and must always be aligned to a page boundary. The physical address of the root page table is stored in the `sptbr` register.

The PTE format for Sv39 is shown in Figure 4.22. Bits 9–0 have the same meaning as for Sv32. Bits 63–48 are reserved for future use and must be zeroed by software for forward compatibility.

We reserved several PTE bits for a possible extension that improves support for sparse address spaces by allowing page-table levels to be skipped, reducing memory usage and TLB refill latency. These reserved bits may also be used to facilitate research experimentation. The cost is reducing the physical address space, but 1 PiB is presently ample. If at some point it no longer suffices, the reserved bits that remain unallocated could be used to expand the physical address space.

Any level of PTE may be a leaf PTE, so in addition to 4 KiB pages, Sv39 supports 2 MiB *megapages* and 1 GiB *gigapages*, each of which must be virtually and physically aligned to a boundary equal to its size.

The algorithm for virtual-to-physical address translation is the same as in Section 4.5.2, except LEVELS equals 3 and PTESIZE equals 8.

4.7 Sv48: Page-Based 48-bit Virtual-Memory System

This section describes a simple paged virtual-memory system designed for RV64 systems, which supports 48-bit virtual address spaces. Sv48 is intended for systems for which a 39-bit virtual address space is insufficient. It closely follows the design of Sv39, simply adding an additional level of page table, and so this chapter only details the differences between the two schemes.

Implementations that support Sv48 should also support Sv39.

We specified two virtual memory systems for RV64 to relieve the tension between providing a large address space and minimizing address-translation cost. For many systems, 512 GiB of virtual-address space is ample, and so Sv39 suffices. Sv48 increases the virtual address space to 256 TiB but increases the physical memory capacity dedicated to page tables, the latency of page-table traversals, and the size of hardware structures that store virtual addresses.

Systems that support Sv48 can also support Sv39 at essentially no cost, and so should do so to support supervisor software that assumes Sv39.

4.7.1 Addressing and Memory Protection

Sv48 implementations support a 48-bit virtual address space, divided into 4 KiB pages. An Sv48 address is partitioned as shown in Figure 4.23. Load and store effective addresses, which are 64 bits, must have bits 63–48 all equal to bit 47, or else an access fault will occur. The 36-bit VPN is translated into a 38-bit PPN via a four-level page table, while the 12-bit page offset is untranslated.

47 39	38 30	29 21	20 12	11 0
VPN[3]	VPN[2]	VPN[1]	VPN[0]	page offset
9	9	9	9	12

Figure 4.23: Sv48 virtual address.

49 39	38 30	29 21	20 12	11 0
PPN[3]	PPN[2]	PPN[1]	PPN[0]	page offset
11	9	9	9	12

Figure 4.24: Sv48 physical address.

63 48	47 37	36 28	27 19	18 10	9 8	7	6	5	4	3	2	1	0
Reserved	PPN[3]	PPN[2]	PPN[1]	PPN[0]	*Res. SW*	D	A	G	U	X	W	R	V
16	11	9	9	9	2	1	1	1	1	1	1	1	1

Figure 4.25: Sv48 page table entry.

The PTE format for Sv48 is shown in Figure 4.25. Bits 9–0 have the same meaning as for Sv32. Any level of PTE may be a leaf PTE, so in addition to 4 KiB pages, Sv48 supports 2 MiB *megapages*, 1 GiB *gigapages*, and 512 GiB *terapages*, each of which must be virtually and physically aligned to a boundary equal to its size.

The algorithm for virtual-to-physical address translation is the same as in Section 4.5.2, except LEVELS equals 4 and PTESIZE equals 8.

Chapter 5

Hypervisor-Level ISA

This chapter is a placeholder for a future RISC-V hypervisor-level common core specification.

The privileged architecture is designed to simplify the use of classic virtualization techniques, where a guest OS is run at user-level, as the few privileged instructions can be easily detected and trapped.

Chapter 6

RISC-V Privileged Instruction Set Listings

This chapter presents instruction set listings for all instructions defined in the RISC-V Privileged Architecture.

31 27 26 25 24 20	19 15	14 12	11 7	6 0	
imm[11:0]	rs1	funct3	rd	opcode	I-type

Trap-Return Instructions

000000000010	00000	000	00000	1110011	URET
000100000010	00000	000	00000	1110011	SRET
001000000010	00000	000	00000	1110011	HRET
001100000010	00000	000	00000	1110011	MRET

Interrupt-Management Instructions

000100000101	00000	000	00000	1110011	WFI

Memory-Management Instructions

000100000100	rs1	000	00000	1110011	SFENCE.VM

Table 6.1: RISC-V Privileged Instructions

Chapter 7

Platform-Level Interrupt Controller (PLIC)

This chapter describes the general architecture for the RISC-V platform-level interrupt controller (PLIC), which prioritizes and distributes global interrupts in a RISC-V system.

7.1 PLIC Overview

Figure 7.1 provides a quick overview of PLIC operation. The PLIC connects global *interrupt sources*, which are usually I/O devices, to *interrupt targets*, which are usually *hart contexts*. The PLIC contains multiple *interrupt gateways*, one per interrupt source, together with a *PLIC core* that performs interrupt prioritization and routing. Global interrupts are sent from their source to an *interrupt gateway* that processes the interrupt signal from each source and sends a single *interrupt request* to the PLIC core, which latches these in the core interrupt pending bits (IP). Each interrupt source is assigned a separate priority. The PLIC core contains a matrix of interrupt enable (IE) bits to select the interrupts that are enabled for each target. The PLIC core forwards an *interrupt notification* to one or more targets if the targets have any pending interrupts enabled, and the priority of the pending interrupts exceeds a per-target threshold. When the target takes the external interrupt, it sends an *interrupt claim* request to retrieve the identifier of the highest-priority global interrupt source pending for that target from the PLIC core, which then clears the corresponding interrupt source pending bit. After the target has serviced the interrupt, it sends the associated interrupt gateway an *interrupt completion* message and the interrupt gateway can now forward another interrupt request for the same source to the PLIC. The rest of this chapter describes each of these components in detail, though many details are necessarily platform specific.

7.2 Interrupt Sources

RISC-V harts can have both local and global interrupt sources. Only global interrupt sources are handled by the PLIC.

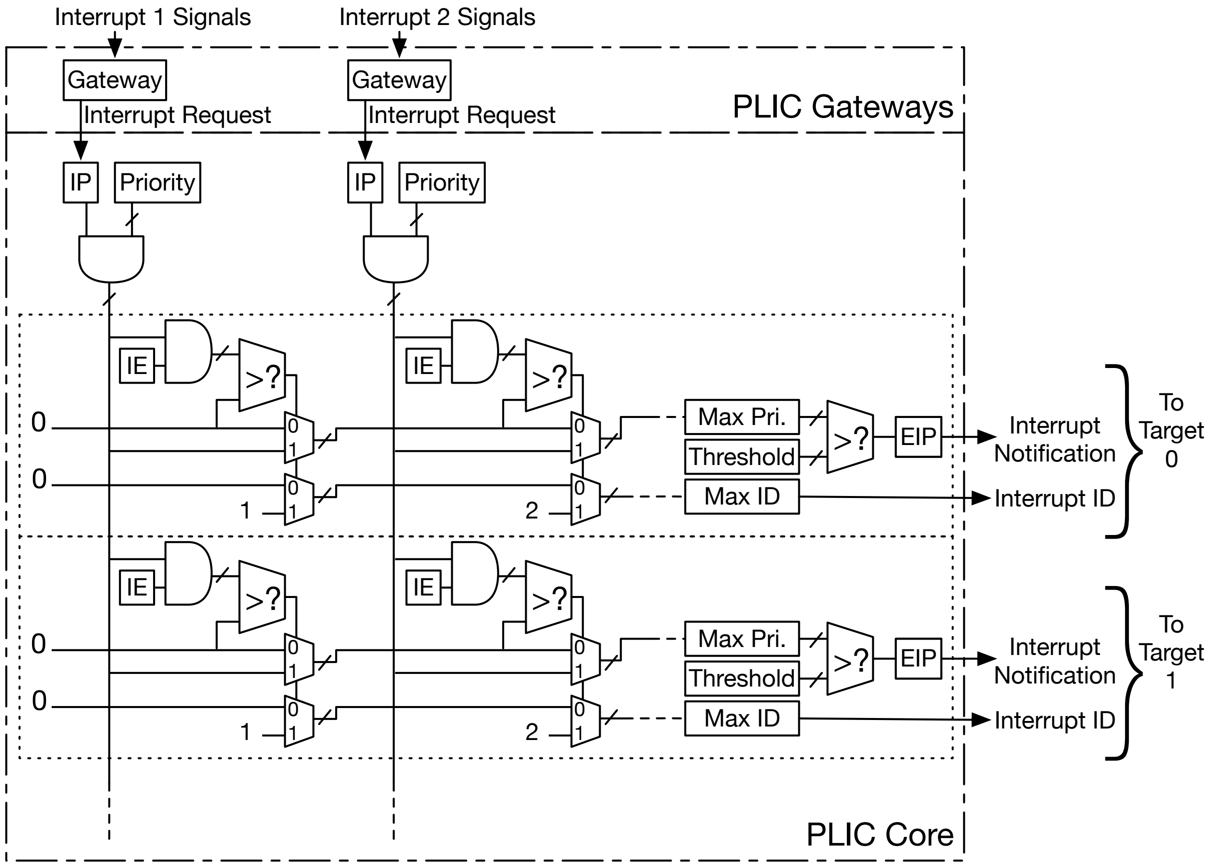

Figure 7.1: Platform-Level Interrupt Controller (PLIC) conceptual block diagram. The figure shows the first two of potentially many interrupt sources, and the first two of potentially many interrupt targets. The figure is just intended to show the logic of the PLIC's operation, not to represent a realistic implementation strategy.

7.2.1 Local Interrupt Sources

Each hart has a number of *local interrupt sources* that do not pass through the PLIC, including the standard software interrupts and timer interrupts for each privilege level. Local interrupts can be serviced quickly since there will be minimal latency between the source and the servicing hart, no arbitration is required to determine which hart will service the request, and the servicing hart can quickly determine the interrupt source using the `mcause` register.

All local interrupts follow a level-based model, where an interrupt is pending if the corresponding bit in `mip` is set. The interrupt handler must clear the hardware condition that is causing the `mip` bit to be set to avoid retaking the interrupt after re-enabling interrupts on exit from the interrupt handler.

Additional non-standard local interrupt sources can be made visible to machine-mode by adding them to the high bits of the `mip`/`mie` registers, with corresponding additional cause values returned in the mcause register. These additional non-standard local interrupts may also be made visible to lower privilege levels, using the corresponding bits in the mideleg register. The priority of non-standard local interrupt sources relative to external, timer, and software interrupts is platform-specific.

7.2.2 Global Interrupt Sources

Global interrupt sources are those that are prioritized and distributed by the PLIC. Depending on the platform-specific PLIC implementation, any global interrupt source could be routed to any hart context.

Global interrupt sources can take many forms, including level-triggered, edge-triggered, and message-signalled. Some sources might queue up a number of interrupt requests. All global interrupt sources are converted to a common interrupt request format for the PLIC.

7.3 Interrupt Targets and Hart Contexts

Interrupt targets are usually hart contexts, where a hart context is a given privilege mode on a given hart (though there are other possible interrupt targets, such as DMA engines). Not all hart contexts need be interrupt targets, in particular, if a processor core does not support delegating external interrupts to lower-privilege modes, then the lower-privilege hart contexts will not be interrupt targets. Interrupt notifications generated by the PLIC appear in the meip/heip/seip/ueip bits of the mip/hip/sip/uip registers for M/H/S/U modes respectively. The notifications only appear in lower-privilege *x*ip registers if external interrupts have been delegated to the lower-privilege modes.

Each processor core must define a policy on how simultaneous active interrupts are taken by multiple hart contexts on the core. For the simple case of a single stack of hart contexts, one for each supported privileged mode, interrupts for higher-privilege contexts can preempt execution of interrupt handlers for lower-privilege contexts. A multithreaded processor core could run multiple independent interrupt handlers on different hart contexts at the same time. A processor core could also provide hart contexts that are only used for interrupt handling to reduce interrupt service latency, and these might preempt interrupt handlers for other harts on the same core.

The PLIC treats each interrupt target independently and does not take into account any interrupt prioritization scheme used by a component that contains multiple interrupt targets. As a result, the PLIC provides no concept of interrupt preemption or nesting so this must be handled by the cores hosting multiple interrupt target contexts.

7.4 Interrupt Gateways

The interrupt gateways are responsible for converting global interrupt signals into a common interrupt request format, and for controlling the flow of interrupt requests to the PLIC core. At most one interrupt request per interrupt source can be pending in the PLIC core at any time, indicated by setting the source's IP bit. The gateway only forwards a new interrupt request to the PLIC core after receiving notification that the interrupt handler servicing the previous interrupt request from the same source has completed.

If the global interrupt source uses level-sensitive interrupts, the gateway will convert the first assertion of the interrupt level into an interrupt request, but thereafter the gateway will not forward an additional interrupt request until it receives an interrupt completion message. On receiving an interrupt completion message, if the interrupt is level-triggered and the interrupt is still asserted, a new interrupt request will be forwarded to the PLIC core. The gateway does not have the facility to retract an interrupt request once forwarded to the PLIC core. If a level-sensitive interrupt source deasserts the interrupt after the PLIC core accepts the request and before the interrupt is serviced, the interrupt request remains present in the IP bit of the PLIC core and will be serviced by a handler, which will then have to determine that the interrupt device no longer requires service.

If the global interrupt source was edge-triggered, the gateway will convert the first matching signal edge into an interrupt request. Depending on the design of the device and the interrupt handler, inbetween sending an interrupt request and receiving notice of its handler's completion, the gateway might either ignore additional matching edges or increment a counter of pending interrupts. In either case, the next interrupt request will not be forwarded to the PLIC core until the previous completion message has been received. If the gateway has a pending interrupt counter, the counter will be decremented when the interrupt request is accepted by the PLIC core.

Unlike dedicated-wire interrupt signals, message-signalled interrupts (MSIs) are sent over the system interconnect via a message packet that describes which interrupt is being asserted. The message is decoded to select an interrupt gateway, and the relevant gateway then handles the MSI similar to an edge-triggered interrupt.

7.5 Interrupt Identifiers (IDs)

Global interrupt sources are assigned small unsigned integer identifiers, beginning at the value 1. An interrupt ID of 0 is reserved to mean "no interrupt".

Interrupt identifiers are also used to break ties when two or more interrupt sources have the same assigned priority. Smaller values of interrupt ID take precedence over larger values of interrupt ID.

7.6 Interrupt Priorities

Interrupt priorities are small unsigned integers, with a platform-specific maximum number of supported levels. The priority value 0 is reserved to mean "never interrupt", and interrupt priority increases with increasing integer values.

Each global interrupt source has an associated interrupt priority held in a platform-specific memory-mapped register. Different interrupt sources need not support the same set of priority values. A valid implementation can hardwire all input priority levels. Interrupt source priority registers should be **WARL** fields to allow software to determine the number and position of read-write bits in each priority specification, if any. To simplify discovery of supported priority values, each priority register must support any combination of values in the bits that are variable within the register, i.e., if there are two variable bits in the register, all four combinations of values in those bits must operate as valid priority levels.

In the degenerate case, all priorities can be hardwired to the value 1, in which case input priorities are effectively determined by interrupt ID.

The supported priority values can be determined as follows: 1) write all zeros to the priority register then 2) read back the value. Any set bits are hardwired to 1. Next, 3) write all ones to the register, and 4) read back the value. Any clear bits are hardwired to 0. Any set bits that were not found to be hardwired in step 2 are variable. The supported priority levels are the set of values obtained by substituting all combinations of ones and zeros in the variable bits within the priority field.

7.7 Interrupt Enables

Each target has a vector of interrupt enable (IE) bits, one per interrupt source. The target will not receive interrupts from sources that are disabled. The IE bits for a single target should be packed together as a bit vector in platform-specific memory-mapped control registers to support rapid context switching of the IE bits for a target. IE bits are **WARL** fields that can be hardwired to either 0 or 1.

A large number of potential IE bits might be hardwired to zero in cases where some interrupt sources can only be routed to a subset of targets.

A larger number of bits might be wired to 1 for an embedded device with fixed interrupt routing. Interrupt priorities, thresholds, and hart-internal interrupt masking provide considerable flexibility in ignoring external interrupts even if a global interrupt source is always enabled.

7.8 Interrupt Priority Thresholds

Each interrupt target has an associated priority threshold, held in a platform-specific memory-mapped register. Only active interrupts that have a priority strictly greater than the threshold will cause a interrupt notification to be sent to the target. Different interrupt targets need not support the same set of priority threshold values. Interrupt target threshold registers should be **WARL** fields to allow software to determine the supported thresholds. A threshold register should always be able to hold the value zero, in which case, no interrupts are masked. If implemented, the threshold register will usually also be able to hold the maximum priority level, in which case all interrupts are masked.

A simple valid implementation is to hardwire the threshold to zero, in which case it has no effect, and the individual enable bits will have to be saved and restored to attain the same effect. While the function of the threshold can be achieved by changing the interrupt-enable bits, manipulating a single threshold value avoids the target having to consider the individual priority levels of each interrupt source, and saving and restoring all the interrupt enables. Changing the threshold quickly might be especially important for systems that move frequently between power states.

7.9 Interrupt Notifications

Each interrupt target has an *external interrupt pending* (EIP) bit in the PLIC core that indicates that the corresponding target has a pending interrupt waiting for service. The value in EIP can change as a result of changes to state in the PLIC core, brought on by interrupt sources, interrupt targets, or other agents manipulating register values in the PLIC. The value in EIP is communicated to the destination target as an interrupt notification. If the target is a RISC-V hart context, the interrupt notifications arrive on the `meip`/`heip`/`seip`/`ueip` bits depending on the privilege level of the hart context.

In simple systems, the interrupt notifications will be simple wires connected to the processor implementing a hart. In more complex platforms, the notifications might be routed as messages across a system interconnect.

The PLIC hardware only supports multicasting of interrupts, such that all enabled targets will receive interrupt notifications for a given active interrupt.

Multicasting provides rapid response since the fastest responder claims the interrupt, but can be wasteful in high-interrupt-rate scenarios if multiple harts take a trap for an interrupt that only one can successfully claim. Software can modulate the PLIC IE bits as part of each interrupt handler to provide alternate policies, such as interrupt affinity or round-robin unicasting.

Depending on the platform architecture and the method used to transport interrupt notifications, these might take some time to be received at the targets. The PLIC is guaranteed to eventually deliver all state changes in EIP to all targets, provided there is no intervening activity in the PLIC core.

The value in an interrupt notification is only guaranteed to hold an EIP value that was valid at some point in the past. In particular, a second target can respond and claim an interrupt while a notification to the first target is still in flight, such that when the first target tries to claim the interrupt it finds it has no active interrupts in the PLIC core.

7.10 Interrupt Claims

Sometime after a target receives an interrupt notification, it might decide to service the interrupt. The target sends an *interrupt claim* message to the PLIC core, which will usually be implemented as a non-idempotent memory-mapped I/O control register read. On receiving a claim message, the PLIC core will atomically determine the ID of the highest-priority pending interrupt for the target and then clear down the corresponding source's IP bit. The PLIC core will then return the ID to the target. The PLIC core will return an ID of zero, if there were no pending interrupts for the target when the claim was serviced.

After the highest-priority pending interrupt is claimed by a target and the corresponding IP bit is cleared, other lower-priority pending interrupts might then become visible to the target, and so the PLIC EIP bit might not be cleared after a claim. The interrupt handler can check the local `meip`/`heip`/`seip`/`ueip` bits before exiting the handler, to allow more efficient service of other interrupts without first restoring the interrupted context and taking another interrupt trap.

It is always legal for a hart to perform a claim even if the EIP is not set. In particular, a hart could set the threshold value to maximum to disable interrupt notifications and instead poll for active interrupts using periodic claim requests, though a simpler approach to implement polling would be to clear the external interrupt enable in the corresponding x`ie` register for privilege mode x.

7.11 Interrupt Completion

After a handler has completed service of an interrupt, the associated gateway must be sent an interrupt completion message, usually as a write to a non-idempotent memory-mapped I/O control register. The gateway will only forward additional interrupts to the PLIC core after receiving the completion message.

7.12 Interrupt Flow

Figure 7.2 shows the messages flowing between agents when handling interrupts via the PLIC.

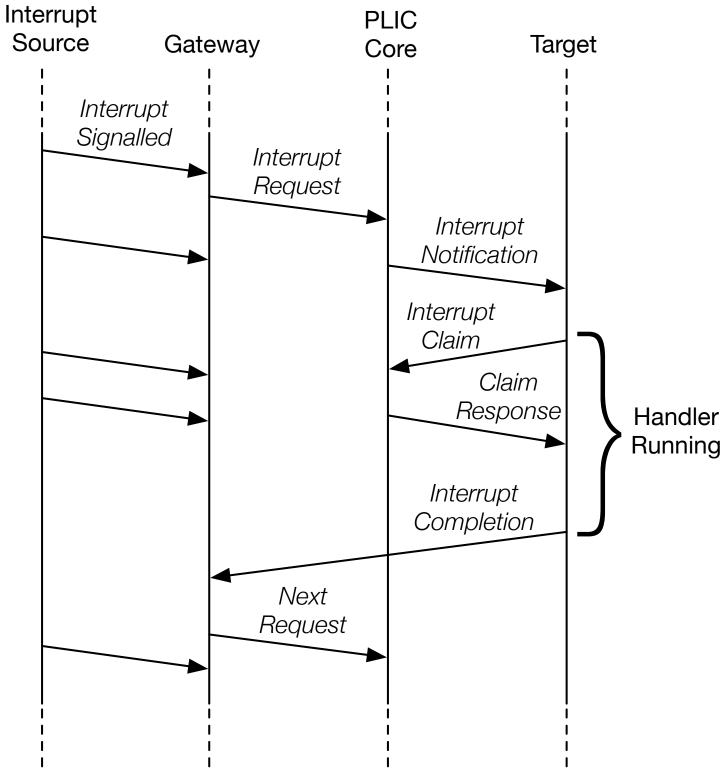

Figure 7.2: Flow of interrupt processing via the PLIC.

The gateway will only forward a single interrupt request at a time to the PLIC, and not forward subsequent interrupts requests until an interrupt completion is received. The PLIC will set the IP bit once it accepts an interrupt request from the gateway, and sometime later forward an interrupt notification to the target. The target might take a while to respond to a new interrupt arriving, but will then send an interrupt claim request to the PLIC core to obtain the interrupt ID. The PLIC core will atomically return the ID and clear the corresponding IP bit, after which no other target can claim the same interrupt request. Once the handler has processed the interrupt, it sends an interrupt completion message to the gateway to allow a new interrupt request.

7.13 PLIC Core Specification

The operation of the PLIC core can be specified as a non-deterministic finite-state machine with input and output messsage queues, with the following atomic actions:

- **Write Register:** A message containing a register write request is dequeued. One of the internal registers is written, where an internal register can be a priority, an interrupt-enable (IE), or a threshold.
- **Accept Request:** If the IP bit corresponding to the interrupt source is clear, a message containing an interrupt request from a gateway is dequeued and the IP bit is set.
- **Process Claim:** An interrupt claim message is dequeued. A claim-response message is enqueued to the requester with the ID of the highest-priority active interrupt for that target, and the IP bit corresponding to this interrupt source is cleared.

The value in the EIP bit is determined as a combinational function of the PLIC Core state. Interrupt notifications are sent via an autonomous process that ensures the EIP value is eventually reflected at the target.

Note that the operation of the interrupt gateways is decoupled from the PLIC core. A gateway can handle parsing of interrupt signals and processing interrupt completion messages concurrently with other operations in the PLIC core.

> *Figure 7.1 is a high-level conceptual view of the PLIC design. The PLIC core can be implemented in many ways provided its behavior can always be understood as following from some sequential ordering of these atomic actions. In particular, the PLIC might process multiple actions in a single clock cycle, or might process each action over many clock cycles.*

7.14 Controlling Access to the PLIC

In the expected use case, only machine mode accesses the source priority, source pending, and target interrupt enables to configure the interrupt subsystem. Lower-privilege modes access these features via ABI, SBI, or HBI calls. The interrupt enables act as a protection mechanism where a target can only signal completion to an interrupt gateway that is currently enabled for that target.

Interrupt handlers that run with lower than machine-mode privilege need only be able to perform a claim read and a completion write, and to set their target threshold value. The memory map for these registers should allow machine mode to protect different targets from each other's accesses, using either physical memory protection or virtual memory page protections.

Chapter 8

Machine Configuration Strings

RISC-V platforms may contain myriad devices, processor cores, and configuration parameters. To support standard software tools across a diversity of platforms, the RISC-V privileged architecture recommends a single, standard textual format to describe the platform and the components therein, including their identity, location in the memory map, and static configuration parameters. This *configuration string* should be made accessible to both external debuggers and to trusted software. The platform configuration string lives in read-only physical memory that must be directly accessible after reset. The platform will describe how to locate this string.

The same configuration string format is used to pass machine configuration information to supervisor-level operating systems, though in general the platform ROM-resident configuration string will be modified and expanded before being passed to a supervisor-level operating system as it might also contain information on external devices attached to chip interfaces and might redact information for resources that are not accessible to supervisor code.

We considered but rejected several alternative existing choices for representing platform resources before settling on a simple character string. Device Tree is a binary encoding designed for 32-bit big-endian systems, a poor match to 64-bit and 128-bit little-endian systems. XML is a general structured data format that has high overhead and is not particularly human-readable. JSON does not support hexadecimal values. Using a simple slightly structured string has many advantages. In particular, it is simple, dense, human-readable, can be processed by standard tools oblivious to the format, and has flexibility to support arbitrary extensions for future needs.

8.1 Syntax

The configuration string is encoded in a human-readable format as a single null-terminated UTF-8 string. The string contains a sequence of key-value pairs. Keys comprise most printable characters (including ASCII 35–58 and 60–122 and printable UTF-8 characters). Values are whitespace-delimited sequences of the following three types: character sequences drawn from the space of legal keys; arbitrary strings enclosed in double-quotes; and nested sequences of key-value pairs enclosed in {braces}. A value sequence is terminated with a semicolon '`;`'.

Quoted strings may contain ASCII characters 32–33 and 35–126 and other printable UTF-8 characters. Literal double-quotes and unprintable bytes can be encoded in quoted strings with the C-style escape sequence `\x`*hh*, e.g., `\x22` for double-quotes or `\x09` for a horizontal tab.

The configuration string does not expressly encode types, as the consumer of a value should know how to interpret it. In particular, the format does not distinguish between numerical types and character sequences.

8.2 Comment Convention

By convention, comments are encoded as key-value pairs with a key of `#`. For example:

```
# this is a comment;
```

8.3 Numeric Encoding Conventions

We recommend a standard encoding for arbitrary-precision numerical types:

1. Decimal integers are encoded as when written in English: as a sequence of the characters 0–9, optionally preceded by a hyphen for negation, e.g. `123` or `-420`. Additionally, underscores may be used to separate digits for clarity, e.g., `12345_67_890` and `1234567890` are equivalent.
2. Decimal reals are encoded like floating-point literals in C source code, e.g., `-1.` or `6.022e23`.
3. Hexadecimal integers are encoded like C integer literals: `0x` followed by a sequence of `0`–`9`, `a`–`f`, and `A`–`F`, optionally preceded by a hyphen for negation, e.g. `0xdeadBEEF` or `-0x1`. Additionally, underscores may be used to separate digits for clarity, e.g., `0x_0001_0000_0000` and `0x100000000` are equivalent.
4. Arbitrary byte sequences may be base-64-encoded by representing every six consecutive bits in the input, in little-endian order, as one byte in the encoded format. Input patterns 0–63 correspond to ASCII 48–111 in the encoding. The pattern is preceded with the prefix `0z`. For example, `abcd` is encoded as `0zQ9fHT1`.
5. Numeric ranges are encoded as two numbers joined with a colon '`:`', where the range is inclusive of the first and last number, e.g., `0x1000:0x1FFF`.

8.4 Platform Key Conventions

A few top-level keys have standard meanings, as shown in Table 8.1.

Key	Description
`core`	Processor core.
`platform`	Platform identification.
`plic`	Platform-Level Interrupt Controller.
`ram`	Read-write memory available after reset.
`rom`	Read-only memory accessible after reset.

Table 8.1: Conventions for top-level platform keys.

The key `interface` is used to indicate a device has a standard interface of the type given in the associated value, e.g., `interface gpio`.

The key `@` is used for address values.

The key `irq` is used for interrupt numbers.

8.5 OS Configuration String Conventions

The string passed to a supervisor-level OS builds upon the platform conventions. Each resource made visible to the OS has a top-level key giving the name of the resource, with a nested interface key-value pair describing the interface to the resource, and an address key with values describing the physical address ranges at which the resource is located.

8.6 Example Configuration String

An example configuration string follows:

```
platform { vendor ucb; arch spike; };
plic {
  interface plic;
  ndevs 2;
  priority { @ 0x40000000:0x40000fff; };
  pending  { @ 0x40001000:0x4000107f; };
  0 {
    0 {
      m {
        ie  { @ 0x40002000:0x4000207f; };
        ctl { @ 0x40200000:0x40200007; };
      };
      s {
        ie  { @ 0x40002080:0x400020ff; };
        ctl { @ 0x40201000:0x40201007; };
      };
    };
  };
};
pcie {
  interface pcie-rv;
  bus {
    @ 0x60000000:0x7fffffff 0x200000000:0x3ffffffff;
    bus 1:63;
  };
  bridge {
    @ 0x50000000:0x53ffffff;
    bus 0;
    irq 0;
  };
};
leds { interface gpio; ngpio 4; @ 0x48002000:0x48002003; };
rtc { @ 0x2000; };
ram { 0 { @ 0x80000000:0x8FFFFFFF; }; };
uart { @ 0x48000000; };
core { 0 { 0 { isa rv64ima; timecmp 0x2008; ipi 0x44000000; }; }; };
```

Chapter 9

Supervisor Binary Interface (SBI)

This chapter is a placeholder to describe the form of the SBIs we're envisioning for the RISC-V supervisor.

The SBI captures the instructions that can be executed together with a set of SBI calls out to the supervisor execution environment (SEE) on a given platform.

Several features that might normally handled by the supervisor operating system (OS) directly are handled via SBI calls to the SEE in RISC-V, including:

- Reset is handled by the SEE and once the machine is set up, the OS kernel is mapped into virtual memory, and its entry point is called.
- Machine-check errors and other non-maskable interrupts are handled by the SEE before vectoring into the OS if recovery is possible.
- Some device drivers may be handled by the SEE, and managed via virtual device calls over the SBI.
- The presence and version of supported instruction-set extensions is obtained via an SBI call to return the configuration string rather than a machine register. This allows for an arbitrarily large definition of instruction set extensions, and simplifies virtualization where the returned machine configuration might be modified to emulate different architectures on a given hardware platform.

The SBI employs the same calling convention as the ABI specified in Volume I of this manual. SBI entry points are located in the uppermost 2 KiB of the virtual address space, so that they may be invoked with a single `jalr` instruction with `x0` as the base register.

Table 9.1 gives a preliminary list of SBI calls.

`const char* sbi_get_config(void);`	Get pointer to configuration string.
`size_t sbi_hart_id(void);`	Get ID of current hart, in range `[0, sbi_num_harts()-1]`.
`int sbi_send_ipi(size_t hart_id);`	Send an interprocessor interrupt; returns 0 on success or -1 if hart ID is invalid.
`bool sbi_clear_ipi(void);`	Clear local interprocessor interrupt. Returns 1 if an IPI was pending, else 0.
`void sbi_shutdown(void);`	Terminate this supervisor-mode process.
`int sbi_console_putchar(uint8_t ch);`	Write byte to debug console (blocking); returns 0 on success, else -1.
`int sbi_console_getchar(void);`	Read byte from debug console; returns the byte on success, or -1 for failure.

`void sbi_remote_sfence_vm(`
`  const uintptr_t* harts, size_t asid);`
	Instruct other harts to execute SFENCE.VM. `harts` points to a bitmask of remote hart IDs; NULL indicates all harts. `asid` holds the address-space ID; 0 indicates all address spaces.

`void sbi_remote_sfence_vm_range(`
`  const uintptr_t* harts, size_t asid,`
`  uintptr_t start, uintptr_t size);`
	Like `sbi_remote_sfence_vm`, but only orders updates to leaf page tables mapping the range `[start, start+size-1]`.

`void sbi_remote_fence_i(`
`  const uintptr_t* harts);`
	Instruct remote harts to execute FENCE.I. `harts` is as with `sbi_remote_sfence_vm`.

`int sbi_mask_interrupt(int which);`	Disable a PLIC interrupt line. Returns 0 if previously disabled, 1 if previously enabled, or -1 if `which` is invalid.
`int sbi_unmask_interrupt(int which);`	Enable a PLIC interrupt line. Return value is as with `sbi_mask_interrupt`.

Table 9.1: SBI calls.

Chapter 10

History

Acknowledgments

Thanks to Allen J. Baum, Ruslan Bukin, Christopher Celio, David Chisnall, Palmer Dabbelt, Dennis Ferguson, Mike Frysinger, Jonathan Neuschäfer, Rishiyur Nikhil, Albert Ou, John Ousterhout, Colin Schmidt, Wesley Terpstra, Matt Thomas, Tommy Thorn, Ray VanDeWalker, and Reinoud Zandijk for feedback on the privileged specification.

10.1 Funding

Development of the RISC-V architecture and implementations has been partially funded by the following sponsors.

- **Par Lab:** Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227). Additional support came from Par Lab affiliates Nokia, NVIDIA, Oracle, and Samsung.
- **Project Isis:** DoE Award DE-SC0003624.
- **ASPIRE Lab**: DARPA PERFECT program, Award HR0011-12-2-0016. DARPA POEM program Award HR0011-11-C-0100. The Center for Future Architectures Research (C-FAR), a STARnet center funded by the Semiconductor Research Corporation. Additional support from ASPIRE industrial sponsor, Intel, and ASPIRE affiliates, Google, Huawei, Nokia, NVIDIA, Oracle, and Samsung.

The content of this paper does not necessarily reflect the position or the policy of the US government and no official endorsement should be inferred.

Bibliography

[1] Robert P. Goldberg. Survey of virtual machine research. *Computer*, 7(6):34–45, June 1974.

[2] Juan Navarro, Sitararn Iyer, Peter Druschel, and Alan Cox. Practical, transparent operating system support for superpages. *SIGOPS Oper. Syst. Rev.*, 36(SI):89–104, December 2002.

[3] Rusty Russell. Virtio: Towards a de-facto standard for virtual I/O devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, July 2008.